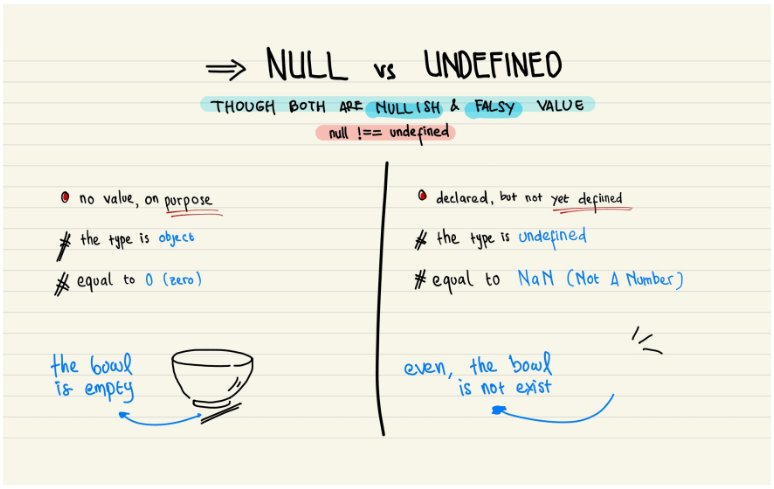
JavaScript Interview Question

1. What is difference between null and undefined in JavaScript ?

In JavaScript, null and undefined are two distinct types that represent different values. By definition, **undefined means a variable has been declared but has not yet been assigned a value, whereas null is an assignment value, meaning that a variable has been declared and given the value of null** .



2. What is an Anonymous function in JavaScript?

It is a function that does not have any name associated with it. Normally we use the *function* keyword before the function name to define a function in JavaScript, however, **in anonymous functions in JavaScript, we use only the *function* keyword without the function name.**

An anonymous function is not accessible after its initial creation, it can only be accessed by a variable it is stored in as a *function as a value*. An anonymous function can also have multiple arguments, but only one expression.

```
function() {
    // Function Body
 }
-------------------------------------------------------------
 ( () => {
    // Function Body...
} )();
-------------------------------------------------------------
 setTimeout(function () {
        console.log("Welcome to GeeksforGeeks!");
    }, 2000);
```

3. What is the difference between == and === in JavaScript?

The main difference between the two operators is how they compare values. The == operator compares the values of two variables after performing type conversion if necessary. On the other hand, the === operator compares the values of two variables without performing type conversion.

```
const num = 10;
const str = "10";
console.log(num == str); // true - The values are the same af
console.log(num === str); // false - The values are different
-------------------------------------------------------------
const bool = true;
const num = 1;
console.log(bool == num); // true - The boolean value true ge
console.log(bool === num); // false - The values are differen
```

4. What is Closure in JavaScript ?

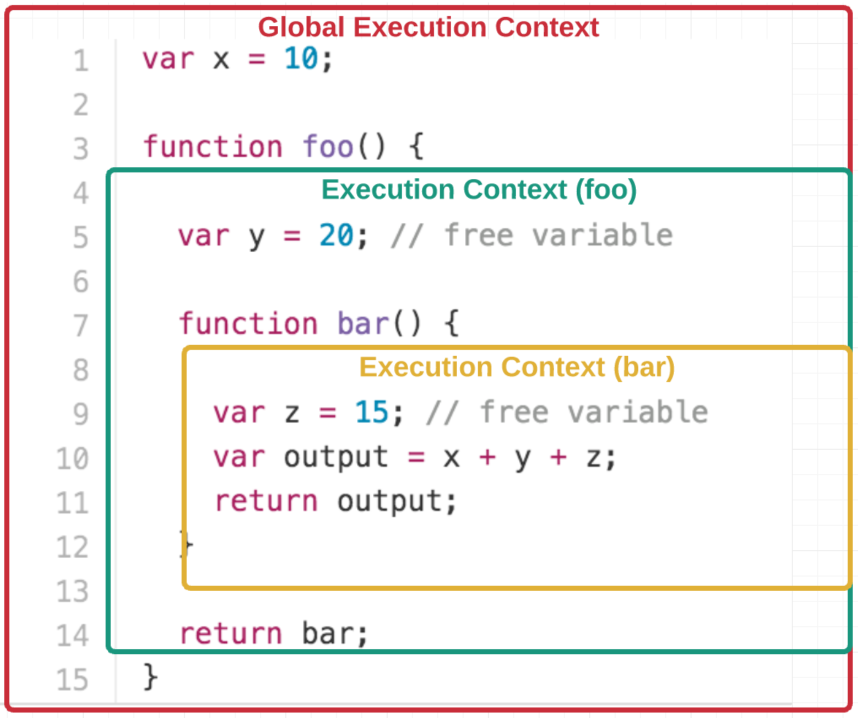
A closure can be defined as a JavaScript feature in which the inner function has access to the outer function variable. In JS every time a closure is created with the creation of a function.

The closure has three scope chains listed as follows:

- Access to its own scope.
- Access to the variables of the outer function.
- Access to the global variables.

Closures are functions that have access to the variables that are present in their scope chain even if the outer function ceases to exist.

To understand this in more detail, let's understand what a scope chain is. Scope chain refers to the fact that **parent scope does not have access to the variables inside its children's scope, but the children's scope does have access to the variables present in its parent scopes**.



```
var x = 10;

function foo() {

  var y = 20; // free variable

  function bar() {

    var z = 15; // free variable
    var output = x + y + z;
    return output;
  }

  return bar;
}
```

How do you create a closure in JavaScript

In JavaScript, a closure is created when an inner function has access to the variables and parameters of its outer function, even after the outer function has finished executing. This allows the inner function to "remember" and access the variables of its outer function, even though the outer function's execution context is no longer active.

```
function outerFunction() {
  let outerVariable = 'I am from the outer function';

  function innerFunction() {
    console.log(outerVariable); // Accessing outerVariable fr
  }

  // Returning the inner function
  return innerFunction;
}

// Calling outerFunction returns innerFunction
const innerFunc = outerFunction();

// Even though outerFunction has finished executing,
// innerFunction still has access to the outerVariable
innerFunc(); // Output: "I am from the outer function"


In this example, innerFunction is a closure because it's defi
outerFunction and has access to outerVariable, which is decla
of outerFunction. When outerFunction is called, it defines in
returns it. Even after outerFunction has finished executing,
retains access to outerVariable, demonstrating the closure be
```

the sequence of execution and how it relates to the call stack:

1. When the script starts executing, the global execution context is created, and the global execution context is pushed onto the call stack.
2. As the script executes, it encounters the line `const innerFunc = outerFunction();`. This line calls the `outerFunction()`.

3. When `outerFunction()` is called, a new execution context for `outerFunction` is created and pushed onto the call stack.
4. Inside `outerFunction`, the variable `outerVariable` is declared and initialized.
5. Then, `innerFunction` is defined inside `outerFunction`, but it's not invoked yet.
6. `outerFunction` returns `innerFunction`, and `outerFunction`'s execution context is popped off the stack. However, since `innerFunction` maintains a reference to `outerVariable`, `outerVariable` is not garbage collected because of the closure.
7. Now, the returned `innerFunction` is assigned to `innerFunc`.
8. Finally, `innerFunc()` is called. This causes the `innerFunction`'s execution context to be created and pushed onto the call stack.
9. Inside `innerFunction`, `console.log(outerVariable)` is executed. Since `innerFunction` has access to the variables of its outer function (`outerFunction`), it can access `outerVariable`.
10. After `console.log(outerVariable)` is executed, `innerFunction`'s execution context is popped off the stack.
11. The call stack becomes empty, and the program execution completes.

Practical applications of Closures

Private Variables and Data Encapsulation:

Closures allow you to create private variables within a function scope that are inaccessible from outside the function. This helps in achieving data encapsulation and preventing unintended modification of variables.

```
function counter() {
    let count = 0;
    return function() {
        return ++count;
    };
}
const increment = counter();
console.log(increment()); // Output: 1
console.log(increment()); // Output: 2
```

Module Pattern:

Closures are often used to implement the module pattern, which allows you to create encapsulated modules with private and public methods and variables.

```
const calculator = (function() {
    let result = 0;
    function add(x) {
        result += x;
    }
    function subtract(x) {
        result -= x;
    }
    function getResult() {
        return result;
    }
    return {
        add,
        subtract,
        getResult
    };
})();
calculator.add(5);
calculator.subtract(3);
console.log(calculator.getResult()); // Output: 2
```

Memoization:

Closures can be used to implement memoization, a technique to cache the results of expensive function calls and reuse them when the same inputs occur again.

```
function memoize(func) {
    const cache = {};
    return function(...args) {
        const key = JSON.stringify(args);
        if (!(key in cache)) {
            cache[key] = func.apply(this, args);
        }
        return cache[key];
    };
}
const factorial = memoize(function(n) {
    if (n === 0 || n === 1) {
        return 1;
    }
    return n * factorial(n - 1);
});
console.log(factorial(5)); // Output: 120
console.log(factorial(5)); // Output: 120 (cached result)
```

Event Handlers:

Closures are commonly used in event handling to maintain the context of an event listener function and access variables from the enclosing scope.

```
function setupCounter() {
    let count = 0;
    document.getElementById('btn').addEventListener('clic
k', function() {
        count++;
        console.log('Count:', count);
    });
}
setupCounter();
```

Currying and Partial Application:

Closures are fundamental to currying and partial application, techniques used to transform a function with multiple arguments into a series of functions each taking a single argument.

```
function add(a) {
    return function(b) {
        return a + b;
    };
}
const add5 = add(5);
console.log(add5(3)); // Output: 8
```

5. let vs. const vs. var

In JS, users can declare a variable using three keywords that are var, let and const.

var

It has the Global Scope or function scoped which means variables defined outside the function can be accessed globally, and variables defined inside a particular function can be accessed within the function.

```
Ex.1 use of the var keyword to declare the variables in JavaS

var a = 10
function f() {
    var b = 20
    console.log(a, b)   // 10, 20
}
f();
console.log(a);  // 10

Ex.2 The behaviour of var variables when declared inside a fu

function f() {
    // It can be accessible any where within this function
    var a = 10;
    console.log(a)   //10
}
f();

// A cannot be accessible outside of function
console.log(a);  //ReferenceError: a is not defined

Ex.3 re-declare a variable with same name in the same scope u
which gives no error in the case of var keyword.

var a = 10
```

```
// User can re-declare variable using var
var a = 8
 // User can update var variable
a = 7
console.log(a);  // 7

Ex.4 code explains the hoisting concept with the var keyword
console.log(a); //undefined
var a = 10;
```

let

It is introduced in the ES6. These variables has the Block scope It can’t be accessible outside the particular code block ({block}).

```
Ex.1 Code explains the block scope of the variables declared

let a = 10;
function f() {
    if (true) {
        let b = 9
        console.log(b);    //9
    }

    // we cant access b outside if block
    console.log(b);   //ReferenceError: b is not defined
}
f()

// It prints 10
console.log(a)  //10

Ex.2 the behaviour of let variables when they are re-declared

let a = 10
```

```
// It is not allowed (re-declartion is not allowed)
let a = 10
// It is allowed (re -intialitzation is allowed)
a = 10

Ex.3 the behaviour of let variables when they are re-declared

let a = 10
if (true) {
    let a = 9 //here it allow to redeclare beacause block sco
    console.log(a) // It prints 9
}
console.log(a) // It prints 10

Ex.4 code explains the hoisting concept with the let variable
console.log(a); //Uncaught ReferenceError: Cannot access 'a'
let a = 10;
```

const

The const keyword has all the properties that are the same as the let, except the user cannot update it and have to assign it with a value at the time of declaration. These variables also have the block scope.

```
Ex.1 code tries to change the value of the const variable.

const a = 10;
function f() {
    a = 9
    console.log(a)  //TypeError:Assignment to constant variab
}
f();

Ex.2
```

.When you use `const` to declare a variable, you cannot reassign the variable to refer to a different value. However, the properties of the object itself can be modified.

```
const a = {
    prop1: 10,
    prop2: 9
};

// It is allowed
a.prop1 = 3;

// It is not allowed
a = {
    b: 10,
    prop2: 9
};
```

var	let	const
The scope of a *var* variable is functional or global scope.	The scope of a *let* variable is block scope.	The scope of a *const* variable is block scope.
It can be updated and re-declared in the same scope.	It can be updated but cannot be re-declared in the same scope.	It can neither be updated or re-declared in any scope.
It can be declared without initialization.	It can be declared without initialization.	It cannot be declared without initialization.
It can be accessed without initialization as its default value is “undefined”.	It cannot be accessed without initialization otherwise it will give ‘referenceError’.	It cannot be accessed without initialization, as it cannot be declared without initialization.
These variables are hoisted.	These variables are hoisted but stay in the temporal dead zone untill the initialization.	These variables are hoisted but stays in the temporal dead zone until the initialization.

6. What is Hoisting in JavaScript ?

Hoisting refers to the process where JavaScript moves Variable and Function declarations to the top of their respective scope during the compilation phase before the actual code execution.

Remember that JavaScript only hoists declarations, not initialization

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

An important distinction to make is that `let` **and** `const` **declarations are also hoisted, but they differ from** `var` **in one significant way. Variables declared with** `let` **or** `const` **are not initialized until their actual position in the code is reached. This is known as the "temporal dead zone."**

```
console.log(myVar); // undefined
var myVar = 5;

console.log(myLet); // ReferenceError: Cannot access 'myLet'
let myLet = 10;
```

7. Explain the difference between function declaration and function expression.

Function declaration and function expression are two ways to define functions in JavaScript.

```
Function Declaration
function add(a, b) {
    return a + b;
}
```

In a function declaration, the keyword `function` is used at the beginning to declare a function. Function declarations are hoisted, meaning they are moved to the top of their containing scope during the compilation phase. Therefore, you can call a function declared using this method before the actual declaration in your code.

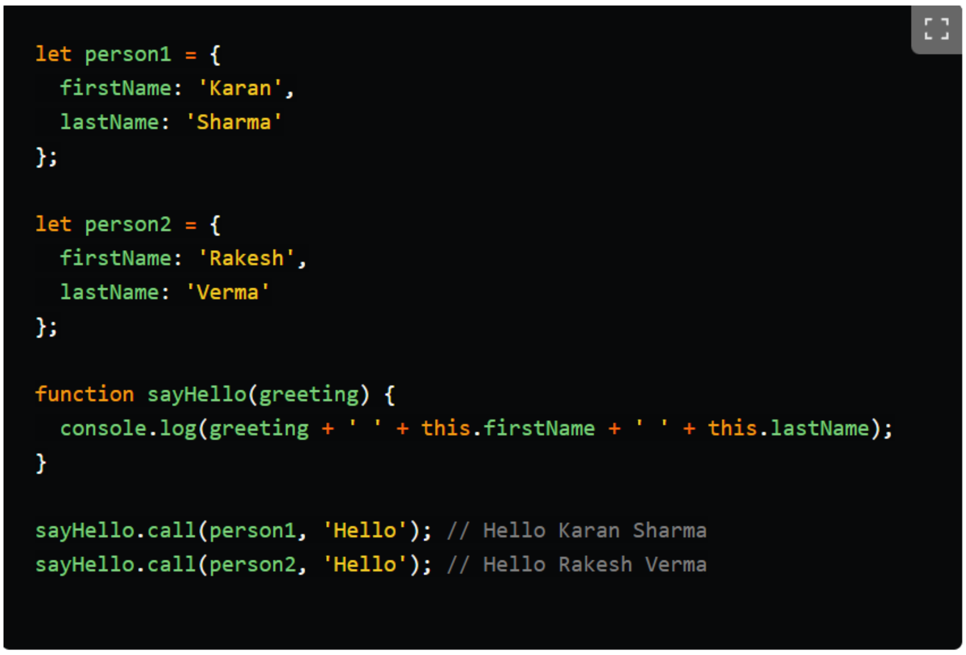
```
Function Expression
var add = function(a, b) {
    return a + b;
};
```

8. What are call, apply and bind?

Call, apply and bind are three **methods that belong to the Function prototype object in JavaScript**. They allow you to change the value of this inside a function and execute it with different arguments.

Call

The call method is used to call a function with a given this value and arguments as comma-separated values.



```
let person1 = {
  firstName: 'Karan',
  lastName: 'Sharma'
};

let person2 = {
  firstName: 'Rakesh',
  lastName: 'Verma'
};

function sayHello(greeting) {
  console.log(greeting + ' ' + this.firstName + ' ' + this.lastName);
}

sayHello.call(person1, 'Hello'); // Hello Karan Sharma
sayHello.call(person2, 'Hello'); // Hello Rakesh Verma
```

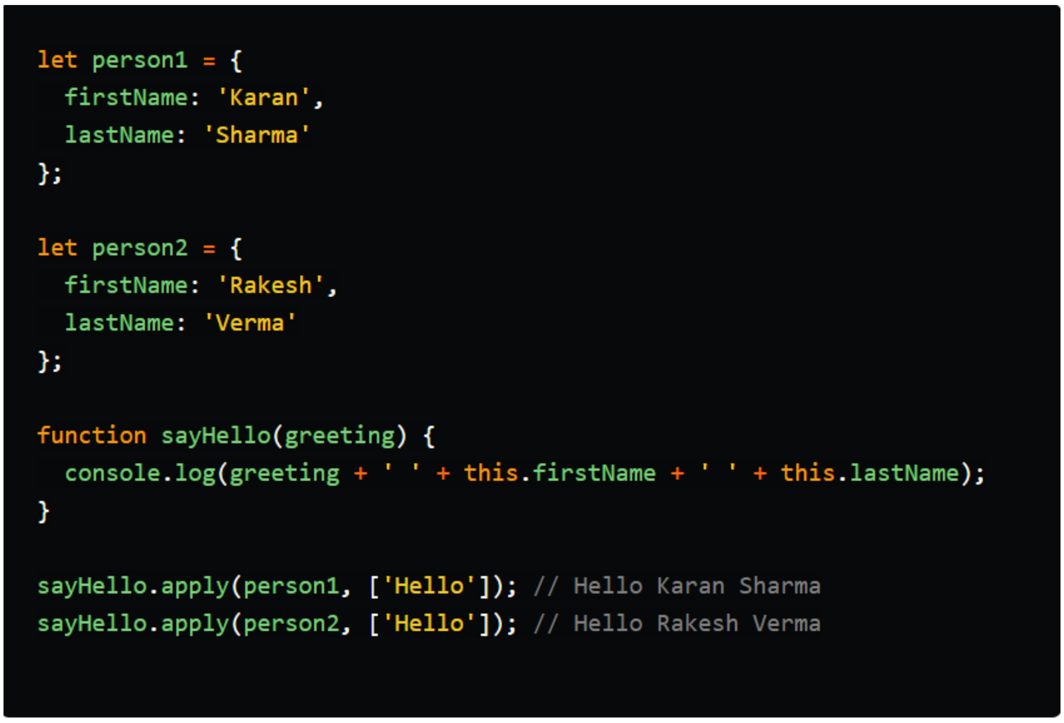
In this example, we use the call method to invoke the sayHello function with the person object as its this value and the string "Hello" as its argument. This way, we can change the context of the function and access the properties of the person object inside the function.

The call method is useful when you want to execute a function with a different this value and a fixed number of arguments. You can also use the

call method to borrow methods from other objects or classes.

Apply

The apply method is very **similar to the call method, except that it takes arguments as an array** (or an array-like object) instead of comma-separated values. For example:

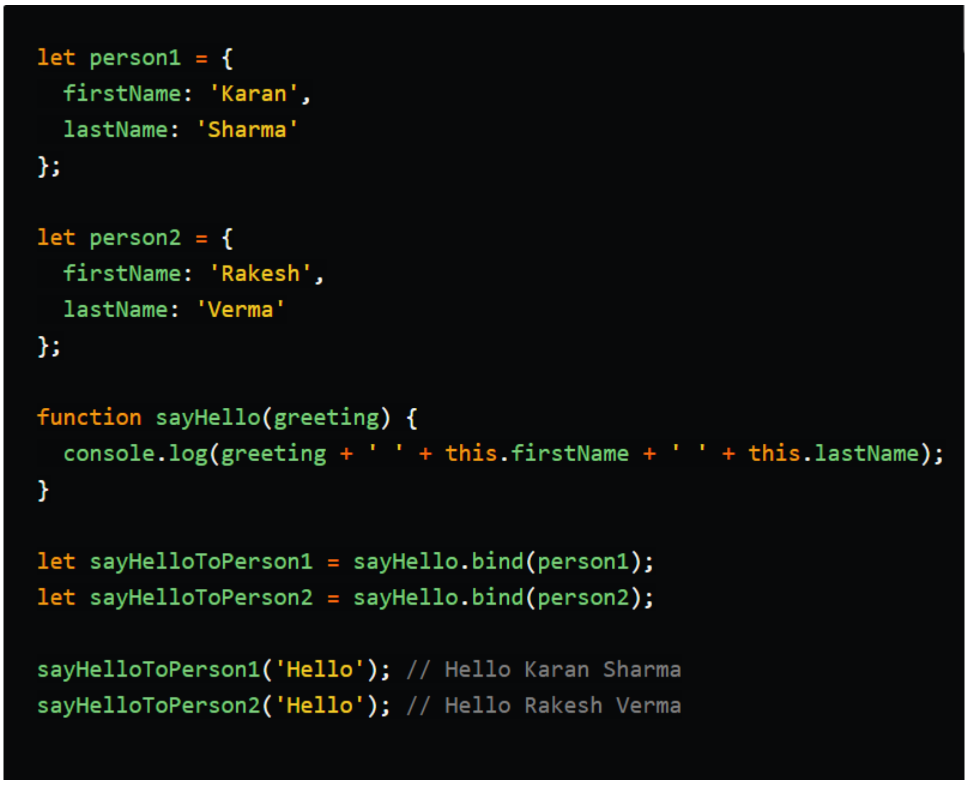


```
let person1 = {
  firstName: 'Karan',
  lastName: 'Sharma'
};

let person2 = {
  firstName: 'Rakesh',
  lastName: 'Verma'
};

function sayHello(greeting) {
  console.log(greeting + ' ' + this.firstName + ' ' + this.lastName);
}

sayHello.apply(person1, ['Hello']); // Hello Karan Sharma
sayHello.apply(person2, ['Hello']); // Hello Rakesh Verma
```

Bind

The bind method is different from the call and apply methods in that it does not execute the function immediately, but returns a new function with a given `this` value and arguments.

By using bind(), you can create new functions with pre-defined contexts, which can then be invoked with the desired arguments.



```javascript
let person1 = {
  firstName: 'Karan',
  lastName: 'Sharma'
};

let person2 = {
  firstName: 'Rakesh',
  lastName: 'Verma'
};

function sayHello(greeting) {
  console.log(greeting + ' ' + this.firstName + ' ' + this.lastName);
}

let sayHelloToPerson1 = sayHello.bind(person1);
let sayHelloToPerson2 = sayHello.bind(person2);

sayHelloToPerson1('Hello'); // Hello Karan Sharma
sayHelloToPerson2('Hello'); // Hello Rakesh Verma
```

9. What is first class function or first order function in JavaScript ?

In JavaScript, a first-order function is a function that is **not** a higher-order function.

A higher-order function is a function that takes one or more functions as arguments, or returns a function as its result.

First-class functions are functions that can be used like any other value, such as being assigned to a variable, passed as an argument to a function or returned as a result from a function.

In JavaScript, **all functions are first-class functions**, so a first-order function is also a first-class function.

10. What is JavaScript ?

- JavaScript is a lightweight, cross-platform, single-threaded, and interpreted compiled programming language.

- It is also known as the scripting language for webpages.
- JavaScript is a weakly typed language or dynamically typed.
- JavaScript can be used for Client-side developments as well as Server-side developments.

11. Is JavaScript Compiled or Interpreted or both ?

JavaScript is both compiled and interpreted. In the earlier versions of JavaScript, it used only the interpreter that executed code line by line and shows the result immediately. But with time the performance became an issue as interpretation is quite slow.

Therefore, in the newer versions of JS, probably after the V8, the JIT compiler was also incorporated to optimize the execution and display the result more quickly. This JIT compiler generates a bytecode that is relatively easier to code. This bytecode is a set of highly optimized instructions.

The V8 engine initially uses an interpreter, to interpret the code. On further executions, the V8 engine finds patterns such as frequently executed functions, and frequently used variables, and compiles them to improve performance.

12. Explain the event loop in JavaScript & Concurrancy Model

We often hear that JavaScript is a single-threaded programming language, which means it executes all of the instructions line by line in a synchronous manner. but there might be cases where any function can take an indeterminate amount of time and our main thread would be blocked, for example calling an API for fetching data from the server-side. In Such case Event loop comes into picture

Memory Organization of JavaScript:

The JavaScript Engine consists of two main components:

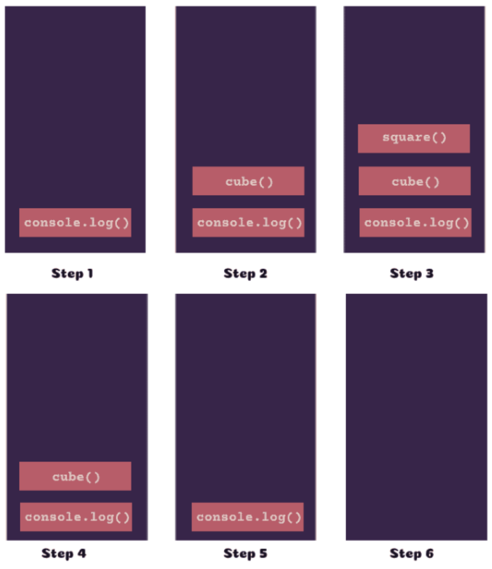
- **Memory Heap** — this is where the memory allocation happens, all of our object variables are assigned here in a random manner.
- **Call Stack** — this is where your function calls are stored.

Understanding the Call Stack

Call stack is a LIFO (last in first out) data structure. All of the function calls are pushed into this call stack

```
function square(b) {
  return b * b ;
}

function cube(x) {
  return x * square(x)
}

console.log(cube(8));
```

1. When we call the `console.log()` line, this `code/function` is pushed into the call stack.
2. Now the above `console.log()` function is called the `cube` function, hence this function is pushed into the stack.
3. After this the `cube` function is calling the `square` function, and now this would be pushed into the stack.
4. Now once the `square` function is executed, and the result is returned, then the `square` function is popped out of the stack.
5. Since at this step, we have got the `square` function result, so `cube` function would be executed and popped out of the call stack.
6. At last, the `console.log()` would be executed and this function would be popped out of the stack and the stack would now be empty.



call stacks are possible to execute the function in a step by step manner, there seems to be no possibility of keeping something into parallel. But there is something called **"WEB APIs"** in the browser environment, which is some additional capabilities provided by browsers in addition to JavaScript Engine.

JavaScript Web APIs

However, since our JavaScript runtime is single-threaded, it can export some tasks to the WEB APIs which helps us to respond to multiple threads. Example of some web APIs are:

- DOM
- Ajax (Network requests)
- setTimeout()

For instance, `setTimeout()` is called, the browser delegates the task to a different thread to calculate the time interval specified in the argument of the `setTimeout()` method, and once done this thread would then call the desired function in callback stack or sent back to callstack.

Event Loop && Event Queue

Event Queue is a special queue, which keeps track of all the functions queues, which are needed to be pushed into the call stack. The event queue is responsible for sending new functions to the track for processing. The queue data structure is required to maintain the correct sequence in which all operations should be sent for execution.

Let's take an example, in the following example, we are using a `setTimeout` function, which will log the "executed" string after 2000 milliseconds.

```
setTimeout(function(){
  console.log("Executed";)
}, 2000);
```

Now when a setTimeout operation is processed in the call stack. On its execution, it calls a web API which fires a timer for 2000 milliseconds. After 2000 milliseconds has been elapsed, the web API, place the callback function of `setTimeout` in the event queue.

Here need to mention that, j
ust placing our function does not necessarily imply that the function will get executed. This function has to be pushed into the call stack for execution and here the event loop comes into the picture. The event loop waits for the function stack to be empty, once the call stack is empty this will push the first function from the event queue to the call stack, and in this way, the desired function will be called.

Thus event loop works in a cyclic manner, where it continually checks whether or not the call stack is empty. If it is empty, new functions are added from the event queue. If it is not, then the current function call is processed.

Referances:

- https://medium.com/@JavaScript-World/javascripts-event-loop-and-concurrency-model-d2a5378e2558
- https://www.loginradius.com/blog/engineering/understanding-event-loop/

JavaScript's Concurrency Model: Single-Threaded but Asynchronous

The concurrency model of JavaScript is based on the concept of an “event loop”. This model enables JavaScript to carry out non-blocking I/O operations despite being a single-threaded language.

13. Immediately Invoked Function Expressions (IIFE) in JavaScript

Immediately Invoked Function Expressions (IIFE) are JavaScript functions that are executed immediately after they are defined. They are typically used to create a local scope for variables to prevent them from polluting the global scope.

The function is wrapped in parentheses `(function() { ... })`, followed by `()` to immediately invoke it.

```
(function() {
    // IIFE code block
    var localVar = 'This is a local variable';
    console.log(localVar); // Output: This is a local variabl
})();
```

IIFEs are commonly used to create private scope in JavaScript, allowing variables and functions to be encapsulated and inaccessible from outside the function.

```
var counter = (function() {
    var count = 0;

    return {
        increment: function() {
            count++;
        },
        decrement: function() {
            count--;
        },
        getCount: function() {
            return count;
        }
    };
})();

// Increment the counter
counter.increment();
counter.increment();
counter.increment();

console.log(counter.getCount()); // Output: 3

// Trying to access the private count variable directly
console.log(counter.count); // Output: undefined (cannot acce

Explanation: Here, count is a private variable scoped to the
The returned object exposes methods (increment, decrement, an
 manipulation and access to the private count variable.
```

Use Cases Of IIFE

- Avoid polluting the global namespace.
- To create **closures in JavaScript**.
- IIFE is used to create private and public variables and methods.
- It is used to execute the **async and await function**.
- It is used to work with **require function**.

14. This keyword in JavaScript ?

In **JavaScript,** `this` **keyword** refers to the **current context or scope** within which code is executing. Its value is determined by how a function is called, and it can dynamically change depending on the invocation context.
The
this keyword refers to different objects depending upon how it is used:

1. Outside of a function body, `this` points to **the global object**.
2. Regular function calls use **the global object** as their execution context.

3. When used by itself, `this` points to the global object.
4. Method calls use **the calling object** as their execution context.
5. How `this` is bound depends on **how** a function is invoked rather than on where it's defined.
6. In a function under strict mode, `this` becomes undefined.
7. During an event, `this` points to the element that triggered the event.
8. Methods such as `call()`, `apply()`, and `bind()` can reassign `this` to any desired object.

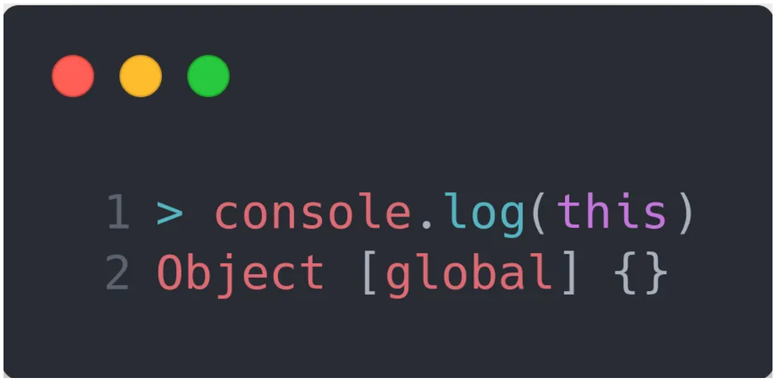
Using this alone

When used alone in JavaScript, outside of any specific context, the behavior of the `this` keyword depends on whether the code is running in strict mode or not.

- In non-strict mode, `this` refers to the global object (e.g., `window` in browsers, `global` in Node.js), representing the global scope.
- In strict mode, `this` is `undefined` when used alone outside of any function or object context. This behavior prevents accidental use of the global object and encourages safer coding practices.

Outside of a function

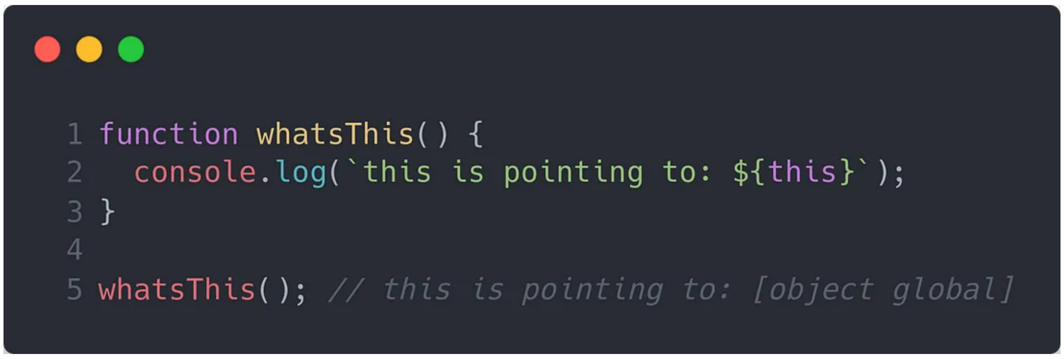
The most important thing to know when determining the value of `this` in JavaScript is that if `this` appears **outside** of the body of a function or method, it points to the global object.



```
> console.log(this)
Object [global] {}
```

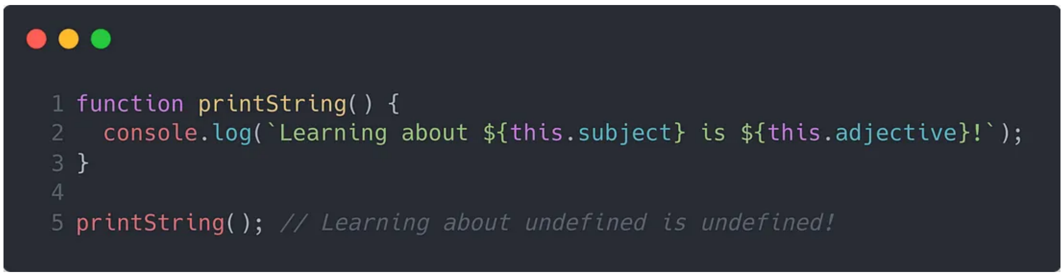
Inside of a function

When you invoke a regular function that contains the `this` keyword in its body, `this` points to the global object. *`this` points to the global object inside a function, but in strict mode it points to `undefined`*.



```javascript
function whatsThis() {
  console.log(`this is pointing to: ${this}`);
}

whatsThis(); // this is pointing to: [object global]
```

In the following code block the function `printString` will look for the properties `subject` and `adjective`



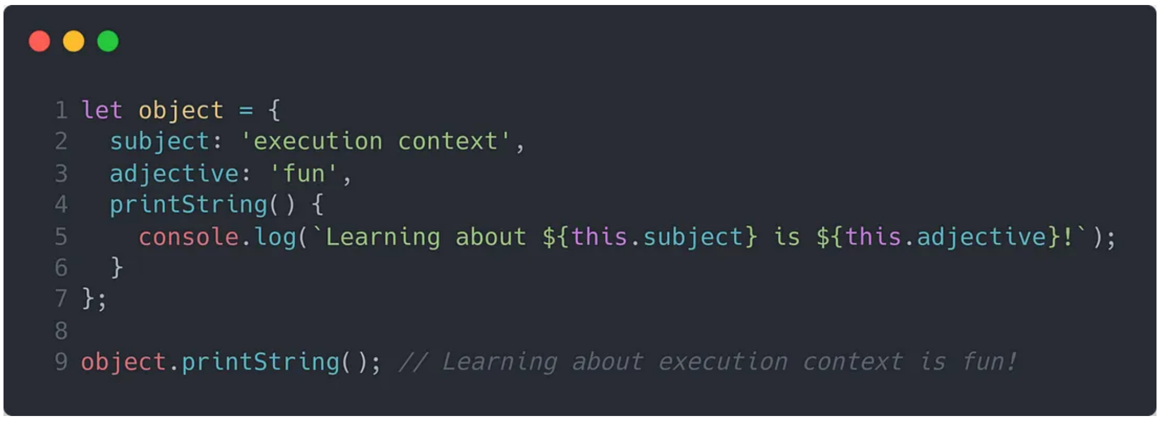
```javascript
function printString() {
  console.log(`Learning about ${this.subject} is ${this.adjective}!`);
}

printString(); // Learning about undefined is undefined!
```

Why did `this.subject` and `this.adjective` evaluate to `undefined`?
As `printString` was invoked as a standalone function, its execution context or `this` binding was implicitly set to the global object.

So `printString` looked at the global object, searched for a property named `subject`, and as there is no property named `subject` on the global object it evaluated to `undefined`. Then the same thing happened with `this.adjective`.

Using this in methods

In the context of an object method in JavaScript, the `this` keyword refers to the object itself, allowing access to its properties and methods within the method's scope.

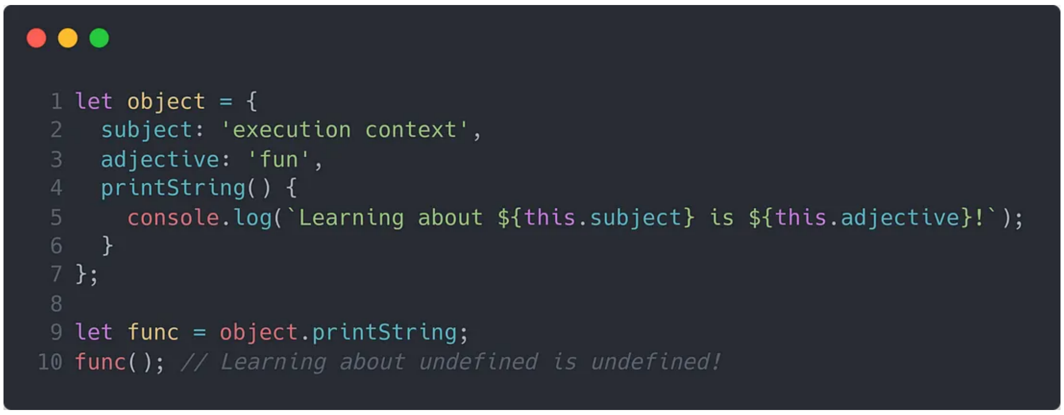


```javascript
let object = {
  subject: 'execution context',
  adjective: 'fun',
  printString() {
    console.log(`Learning about ${this.subject} is ${this.adjective}!`);
  }
};

object.printString(); // Learning about execution context is fun!
```

In this example, the method `printString` looks for two properties, `subject` and `adjective`. How does it know where to look for them?

You may think that it just looks for those properties in the same object that the method appears in. That is not quite accurate though. Instead, it looks for the properties **in the object that calls the method**.

That is a very important distinction. What would happen if we stored `object.printString` in a variable and then called it?



```javascript
 1 let object = {
 2   subject: 'execution context',
 3   adjective: 'fun',
 4   printString() {
 5     console.log(`Learning about ${this.subject} is ${this.adjective}!`);
 6   }
 7 };
 8
 9 let func = object.printString;
10 func(); // Learning about undefined is undefined!
```

Notice that we have stored `object.printString` in a variable called `func` and then invoked `func` on line 10. Now suddenly `this` is no longer pointing to `object`. Can you guess where it is pointing?

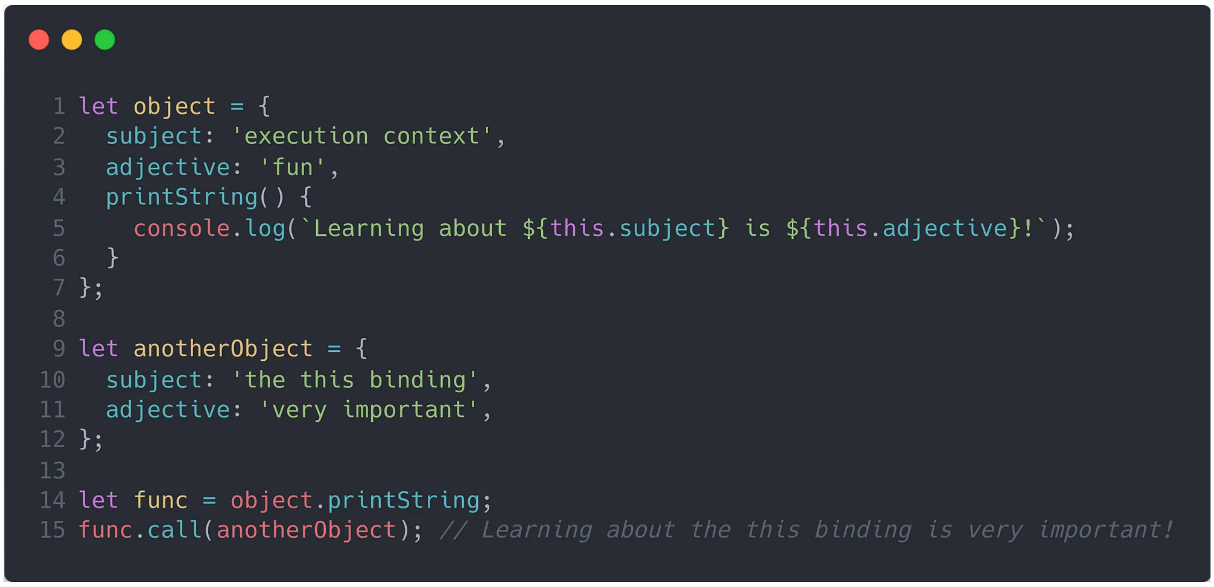
If you said the global object, you were correct. `func` is called as a standalone function on line 10 and not as a method, and as we learned earlier the implicit execution context of a function is the global object.

This teaches us something very important about the binding of `this`. The value of `this` is determined, not by where it appears in your code, but by **how** a function or method is invoked

Explicitly setting the context using call

So far we have mentioned several times how `this` is **implicitly** set when you invoke a function or method. That implies that we can also **explicitly** set the execution context.

All functions have access to a property method named `call` that they inherit from `Function.prototype`. We can invoke the `call` method on a function and explicitly set the execution context to whatever object we pass in as the first argument.



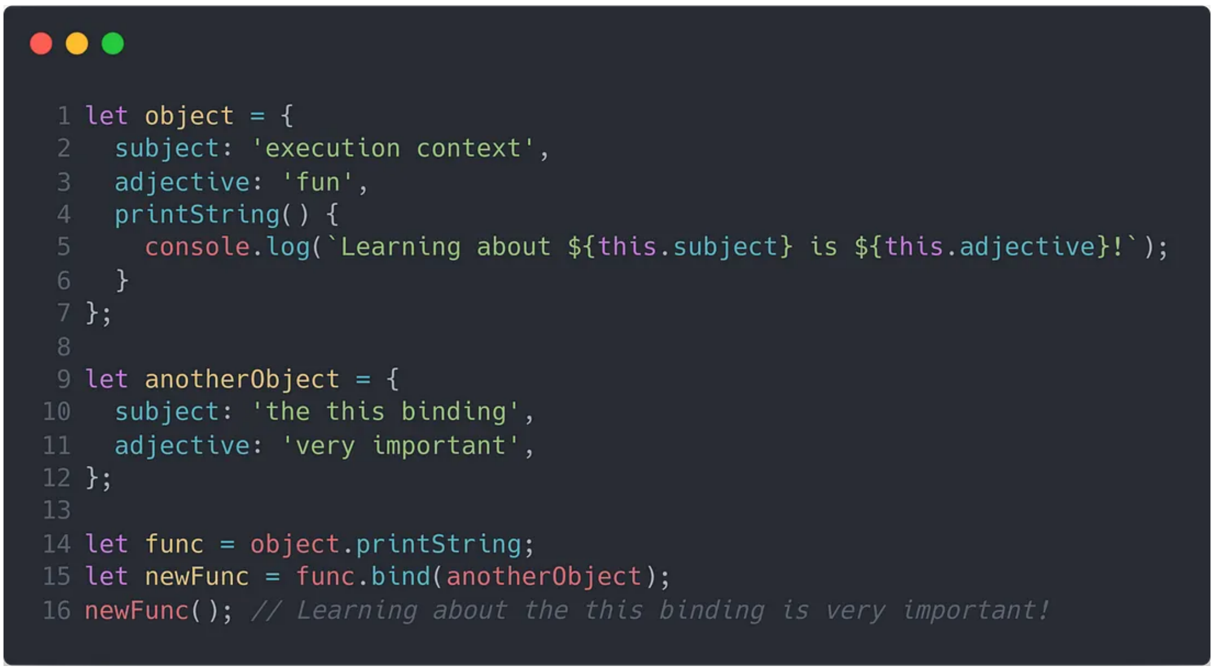
```
 1 let object = {
 2   subject: 'execution context',
 3   adjective: 'fun',
 4   printString() {
 5     console.log(`Learning about ${this.subject} is ${this.adjective}!`);
 6   }
 7 };
 8
 9 let anotherObject = {
10   subject: 'the this binding',
11   adjective: 'very important',
12 };
13
14 let func = object.printString;
15 func.call(anotherObject); // Learning about the this binding is very important!
```

Notice that we invoke the func function on line 15 with the call method, and we pass in anotherObject as an argument. Now the execution context of func on line 15 is bound to the object anotherObject, and so it looks there for the properties subject and adjective.

Note that using call does not mutate func, so we can still call func as a standalone function and its execution context will still be implicitly bound to the global object.

Permanently binding an object

All functions also have access to a property method named bind that they inherit from Function.prototype. bind works differently from call in that it doesn't invoke the function. Instead, it returns a new function with its execution context permanently bound to the passed in argument.



```javascript
let object = {
  subject: 'execution context',
  adjective: 'fun',
  printString() {
    console.log(`Learning about ${this.subject} is ${this.adjective}!`);
  }
};

let anotherObject = {
  subject: 'the this binding',
  adjective: 'very important',
};

let func = object.printString;
let newFunc = func.bind(anotherObject);
newFunc(); // Learning about the this binding is very important!
```

When we invoke newFunc( ), we see that its execution context has been bound to anotherObject. It's no longer possible to change the execution context of newFunc as it has been permanently bound to anotherObject. Note that func has not been mutated.

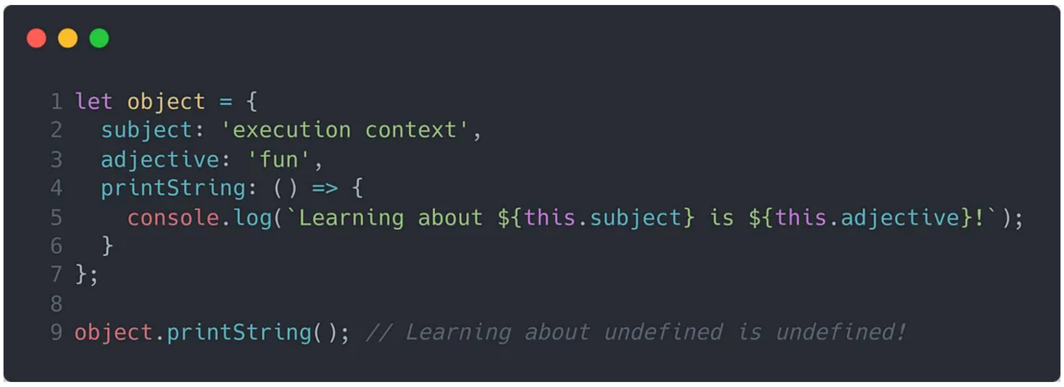
Summary:

- We can explicitly set the execution context using `call` or `bind`.
- `call` invokes the function with its execution context set to the passed in argument.

- `bind` returns a **new** function with the execution context permanently bound to the passed in argument.

Why arrow functions are special

When invoked, this.subject and this.adjective evaluate to undefined. This is because arrow functions do not have their own this binding, instead, **they establish the binding of this from the surrounding code.**



```javascript
let object = {
  subject: 'execution context',
  adjective: 'fun',
  printString: () => {
    console.log(`Learning about ${this.subject} is ${this.adjective}!`);
  }
};

object.printString(); // Learning about undefined is undefined!
```

Once an arrow function is defined in your code, its execution context cannot be changed. If the arrow funtion is defined within the body of another function, it will always have the same context as the outer function.

Unlike with non-arrow functions, the execution context for arrow functions is determined lexically, which means that we determine the context from where the arrow function is defined rather than from how it is invoked.

15. How do you check the type of a variable in JavaScript?

In JavaScript, **typeof operator** returns the data type of its operand in the form of a string. The operand can be any object, function, or variable.

```
// Define variables with different primitive data types
const num = 10;
const str = "Hello";
const bool = true;
const undef = undefined;
const nul = null;
const sym = Symbol("symbol");
const bigInt = 9007199254740991n;

// Use typeof operator to determine the data type
console.log(typeof num);   // Output: "number"
console.log(typeof str);   // Output: "string"
console.log(typeof bool);  // Output: "boolean"
console.log(typeof undef); // Output: "undefined"
console.log(typeof nul);   // Output: "object" (typeof null i
                           // it returns "object")
console.log(typeof sym);   // Output: "symbol"
console.log(typeof bigInt);// Output: "bigint"
```

What is a callback function?

A callback is a function that is passed as an argument to another function, and is called after the main function has finished its execution. The main function is called with a callback function as its argument, and when the main function is finished, it calls the callback function to provide a result. Callbacks allow you to handle the results of an asynchronous operation in a non-blocking manner, which means that the program can continue to run while the operation is being executed.

```


function mainFunction(callback) {
  console.log("Performing operation...");
  // Use setTimeout to simulate an asynchronous operation
  setTimeout(function() {
    callback("Operation complete");
  }, 1000);
}

// Define the callback function
function callbackFunction(result) {
  console.log("Result: " + result);
}

// Call the main function with the callback function
mainFunction(callbackFunction);
Output:
Performing operation...
Result: Operation complete

1. We first define a mainFunction that takes a callback as an
2. The mainFunction uses setTimeout to simulate an asynchrono
3. The setTimeout function takes two arguments: a callback fu
4. The setTimeout function calls the callback function with t
result of the operation after the specified delay time.
5. We then define a callbackFunction that logs the result of
6. Finally, we call the mainFunction with the callbackFunctio
```

How can you avoid callback hell in JavaScript

Callback hell refers to the situation in JavaScript where multiple nested callbacks are used, leading to code that is difficult to read, understand, and maintain. This often occurs in asynchronous code, such as when dealing with multiple asynchronous operations like network requests or file I/O.

```
fetchData(function(err, data) {
    if (err) {
        console.error('Error:', err);
    } else {
        console.log('Data:', data);
        processData(data, function(err, result) {
            if (err) {
                console.error('Error:', err);
            } else {
                console.log('Processed Result:', result);
            }
        });
    }
});
```

Use Promises:

- Promises provide a more structured way to handle asynchronous operations and avoid deeply nested callbacks.
- Promises allow you to chain asynchronous operations using `.then()` and handle errors with `.catch()`.

```
fetchData()
    .then(data => {
        console.log('Data:', data);
        return processData(data);
    })
    .then(result => {
        console.log('Processed Result:', result);
    })
    .catch(error => {
        console.error('Error:', error);
    });
```

1. **Use Async/Await:**
    - Async/await is a modern JavaScript feature that provides a more synchronous-looking syntax for handling asynchronous operations.
    - It allows you to write asynchronous code in a linear, sequential style without explicit callbacks or chaining promises.
    - Async functions return a Promise, making them compatible with Promise-based APIs.

```
async function fetchData() {
    // Simulate asynchronous operation
    return new Promise(resolve => {
        setTimeout(() => {
            const data = 'Some data';
            resolve(data);
        }, 1000);
    });
}

async function processData() {
    const data = await fetchData();
    console.log('Data:', data);
    // Simulate further processing
    return 'Processed result';
}

(async () => {
    try {
        const result = await processData();
        console.log('Processed Result:', result);
    } catch (error) {
        console.error('Error:', error);
    }
})();
```

1. **Modularize Code:**
    - Break down your code into smaller, more manageable functions.
    - Each function should ideally perform a single task, making the code easier to understand and maintain.
    - This approach also encourages code reuse and improves testability.
2. **Avoid Pyramid Structure:**
    - Refactor your code to flatten the nested structure by breaking down complex operations into smaller, more manageable pieces.
    - Avoid deeply nested callbacks by extracting them into separate functions.

Explain the concept of a pure function and Impure Function

In JavaScript, a pure function is a function that always returns the same output for the same input and does not cause any side effects.

In other words, it only depends on its input parameters and does not modify any external state or global variables.

Pure functions are deterministic, meaning that they will always produce the same output given the same input, which makes them easy to test, maintain, and reason about

```
Pure Function
function add(a, b) {
 return a + b;
}
```

an impure function is a function that can cause side effects and modify external state or global variables.

Examples of impure functions include functions that modify the DOM, interact with external APIs or databases, and generate random numbers.

```
let counter = 0

function increment() {
 counter++;
 console.log(counter);
};
```

Explain the concept of function currying.

Function currying is a concept in functional programming where a function with multiple parameters is transformed into a sequence of functions, each taking only one parameter.

We simply wrap a function inside a function, which means we are going to return a function from another function to obtain this kind of translation. The parent function takes the first provided argument and returns the function that takes the next argument and this keeps on repeating till the number of arguments ends. Hopefully, the function that receives the last argument returns the expected result.

```
// Non-curried function
function add(a, b, c) {
  return a + b + c;
}

console.log(add(2, 3, 4)); // Output: 9

// Curried version of the function
function curriedAdd(a) {
  return function(b) {
    return function(c) {
      return a + b + c;
    };
  };
}

console.log(curriedAdd(2)(3)(4)); // Output: 9
```

Why is currying useful in JavaScript?

- It helps us to create a higher-order function
- It reduces the chances of error in our function by dividing it into multiple smaller functions that can handle one responsibility.
- It is very useful in building modular and reusable code
- It helps us to avoid passing the same variable multiple times
- It makes the code more readable

21. What is prototypal inheritance?

Prototypal inheritance is a fundamental concept in JavaScript, and it's how objects inherit properties and methods from other objects. Unlike classical inheritance found in languages like Java or C++, where classes are used to create objects and inheritance hierarchies, JavaScript uses prototype-based inheritance.

In JavaScript, each object has a private property called `[[Prototype]]` (also referred to as `__proto__`), which points to another object. When you access a property or method on an object, and it doesn't exist on the object itself, JavaScript looks up the prototype chain until it finds the property or method or reaches the end of the chain.

```
// Parent object (prototype)
const animal = {
  type: 'Animal',
  sound: function() {
    console.log('Some sound');
  }
};

// Child object
const dog = {
  breed: 'Golden Retriever'
};

// Setting the prototype of dog to animal using __proto__
dog.__proto__ = animal;

// Accessing properties and methods via prototypal inheritanc
console.log(dog.type); // Output: Animal
dog.sound(); // Output: Some sound




// Parent object (prototype)
const animal = {
  type: 'Animal',
  sound: function() {
    console.log('Some sound');
  }
};

// Child object
const dog = {
  breed: 'Golden Retriever'
};

// Setting the prototype of dog to animal
Object.setPrototypeOf(dog, animal);

// Accessing properties and methods via prototypal inheritanc
console.log(dog.type); // Output: Animal
dog.sound(); // Output: Some sound
```

prototype vs [[prototype]]

1. `prototype` **Property**:
    - The `prototype` property is a property of constructor functions in JavaScript.
    - It's used to define properties and methods that will be inherited by instances created with that constructor function.
    - When you define a method or property on a constructor's `prototype`, it becomes accessible to instances created with that constructor.
    - It's an explicit property that you can access and modify directly.

```
function Person(name) {
  this.name = name;
}
Person.prototype.sayHello = function() {
  console.log("Hello, my name is " + this.name);
};

const john = new Person("John");
john.sayHello(); // Output: "Hello, my name is John"
```

1. `[[Prototype]]` **Property (or** `__proto__`**):**
    - The `[[Prototype]]` property is an internal property of JavaScript objects.
    - It's a reference to the prototype object of the object.
    - It represents the inheritance relationship between objects.
    - When you access a property or method on an object, and it's not found on the object itself, JavaScript looks up the prototype chain by following the `[[Prototype]]` chain until it finds the property or reaches the end of the chain.
    - It's accessed indirectly through the `__proto__` property or the `Object.getPrototypeOf()` method.

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

const student = {};
student.__proto__ = person;

console.log(student.name); // Output: "John"
```

How do you create an object in JavaScript?

In JavaScript, you can create objects using various methods, including object literals, constructor functions, and the `Object.create()` method. Here are examples of each method:

1. **Object Literals:**
    - The simplest way to create an object is by using object literals, which are enclosed in curly braces `{}`.

```
// Object literal
const person = {
  name: 'John',
  age: 30,
  greet: function() {
    console.log('Hello!');
  }
};
```

1. **Constructor Functions**:
   - Constructor functions are traditional functions used with the `new` keyword to create objects. Inside the constructor function, you use `this` to refer to the object being created.

```
// Constructor function
function Person(name, age) {
  this.name = name;
  this.age = age;
  this.greet = function() {
    console.log('Hello!');
  };
}

// Creating an object using the constructor function
const john = new Person('John', 30);
```

1. **Object.create()**:
   - The `Object.create()` method creates a new object with the specified prototype object and properties.

```
// Prototype object
const personPrototype = {
  greet: function() {
    console.log('Hello!');
  }
};

// Creating an object using Object.create()
const john = Object.create(personPrototype);
john.name = 'John';
john.age = 30;
```

1. **Class Syntax** (ES6+):
    - With the introduction of classes in ES6, you can use class syntax to create objects, which is syntactic sugar over constructor functions.

```
// Class syntax
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  greet() {
    console.log('Hello!');
  }
}

// Creating an object using the class syntax
const john = new Person('John', 30);
```

Explain the difference between Object.create and the constructor pattern.

Constructor Pattern:

- In the constructor pattern, you define a function (constructor function) that serves as a blueprint for creating objects.
- Objects are created using the `new` keyword followed by the constructor function.
- Inside the constructor function, you use the `this` keyword to refer to the newly created object, and you attach properties and methods to it.

```
function Person(name, age) {
    this.name = name;
    this.age = age;
}
var person1 = new Person('John', 30);
```

Object.create:

- `Object.create` is a method that creates a new object, using an existing object as the prototype of the newly created object.
- It allows you to directly specify the prototype object that you want the new object to inherit from.
- You can optionally pass a second argument to `Object.create` to define properties for the new object.

```
var personProto = {
    greet: function() {
        return 'Hello, my name is ' + this.name;
    }
};
var person1 = Object.create(personProto);
person1.name = 'John';
person1.age = 30;
```

```
// Creating an object
const person = {
  name: 'John',
  age: 30,
  greet: function() {
    console.log(`Hello, my name is ${this.name} and I am ${th
  }
};
Accessing Object Properties:
Object.keys(obj): Returns an array of own enumerable property
Object.values(obj): Returns an array of own enumerable proper
Object.entries(obj): Returns an array of own enumerable [key,

const keys = Object.keys(person);
console.log('Keys:', keys);
// Output: ["name", "age", "greet"]

const values = Object.values(person);
console.log('Values:', values);
```

```
// Output: ["John", 30, [Function: greet]]

const entries = Object.entries(person);
console.log('Entries:', entries);
// Output: [["name", "John"], ["age", 30], ["greet", [Functio


Cloning and Merging Objects:
Object.assign(target, source1, source2, ...):
Copies enumerable own properties from source objects to the t
nd returns the target object.
Object.create(proto[, propertiesObject]): Creates a new objec
prototype object and properties.

const clone = Object.assign({}, person);
console.log('Clone:', clone);
// Output: { name: "John", age: 30, greet: [Function: greet]

const newObj = { city: 'New York' };
const merged = Object.assign({}, person, newObj);
console.log('Merged:', merged);
// Output: { name: "John", age: 30, greet: [Function: greet],

Object Mutability:
Object.freeze(obj): Freezes an object, making it immutable
(properties cannot be added, modified, or deleted).
Object.seal(obj): Prevents new properties from being added to
object and prevents existing properties from being removed.
Existing properties can still be modified.
Object.preventExtensions(obj): Prevents new properties from b
but allows existing properties to be modified or deleted.

const frozen = Object.freeze(person);
frozen.name = 'Jane'; // Attempting to modify a frozen object
console.log('Frozen:', frozen); // Output: { name: "John", ag

const sealed = Object.seal(person);
sealed.name = 'Jane'; // Modifying a sealed object
sealed.gender = 'Male'; // Attempting to add a new property t
console.log('Sealed:', sealed); // Output: { name: "Jane", ag

const prevented = Object.preventExtensions(person);
prevented.name = 'Jane'; // Modifying a prevented object
prevented.gender = 'Male'; // Attempting to add a new propert
console.log('Prevented:', prevented); // Output: { name: "Jan

Property Descriptor:
Object.getOwnPropertyDescriptor(obj, prop): Returns a propert
for the given property of obj.
Object.defineProperty(obj, prop, descriptor): Defines a new p
 on obj, or modifies an existing property, and returns the ob
const descriptor = Object.getOwnPropertyDescriptor(person, 'n
console.log('Descriptor:', descriptor); // Output: { value: "

Checking Object Properties:
Object.hasOwnProperty(prop): Returns a boolean indicating whe
the object has the specified property as its own property (no
console.log('Has own property "name":', person.hasOwnProperty
console.log('Has own property "city":', person.hasOwnProperty

Conversion to String:
Object.toString(obj): Returns a string representation of obj.
const str = Object.toString(person);
console.log('String representation:', str); // Output: [objec
```

In JavaScript, a Map is a built-in data structure that allows you to store key-value pairs where both the keys and values can be of any data type, including objects and primitive values. Unlike objects, Maps maintain the insertion order of elements, making them useful for scenarios where the order of elements is important.

```
// Creating a Map using the Map constructor
const myMap = new Map();
```

```
// Adding key-value pairs to the Map
myMap.set('key1', 'value1');
myMap.set('key2', 'value2');
myMap.set('key3', 'value3');


// Creating a Map using Map literal syntax
const myMap = new Map([
  ['key1', 'value1'],
  ['key2', 'value2'],
  ['key3', 'value3']
]);
```

.

```
// Creating a Map
const myMap = new Map();

// Adding key-value pairs with different data types
myMap.set('name', 'John Doe'); // String key, String value
myMap.set(123, 'Age'); // Number key, String value
myMap.set(true, { city: 'New York', country: 'USA' }); // Boo
myMap.set(Symbol('id'), ['apple', 'banana', 'orange']); // Sy

// Accessing values from the Map
console.log(myMap.get('name')); // Output: John Doe
console.log(myMap.get(123)); // Output: Age
console.log(myMap.get(true)); // Output: { city: 'New York',
console.log(myMap.get(Symbol('id'))); // Output: undefined
(Symbols are unique, so this returns undefined)

// Checking if keys exist in the Map
console.log(myMap.has('name')); // Output: true
console.log(myMap.has(456)); // Output: false

// Deleting a key-value pair from the Map
myMap.delete(true);
```

```javascript
// Iterating over the key-value pairs in the Map
myMap.forEach((value, key) => {
  console.log(`${key}: ${JSON.stringify(value)}`);
});

// Clearing all key-value pairs from the Map
myMap.clear();
```

Objects and Maps are both used to store collections of data in JavaScript, but they have some differences in terms of their features and use cases.

Objects:

1. **Key-Value Pairs**: Objects are collections of key-value pairs, where each key is unique. Keys are typically strings or symbols, and values can be any data type.
2. **Property Access**: You can access properties of an object using dot notation (`obj.property`) or bracket notation (`obj['property']`).
3. **Prototype Chain**: Objects in JavaScript have a prototype chain, which allows them to inherit properties and methods from other objects through prototypal inheritance.
4. **Literal Syntax**: Objects can be created using object literal syntax `{}` or constructor functions.

Maps:

1. **Arbitrary Keys**: Maps allow for keys of any data type, including objects, arrays, and functions. Keys are not coerced to strings like in objects.
2. **Order Preservation**: Maps maintain the order of key-value pairs, whereas objects do not guarantee order (though in practice, modern JavaScript engines preserve the order of properties in most cases).
3. **Iterable**: Maps are iterable, which means you can loop over them using loops like `for...of` or methods like `forEach()`.
4. **Size Property**: Maps have a `size` property that returns the number of key-value pairs in the Map.

5. **Better for Performance**: Maps can offer better performance for scenarios involving frequent additions and removals of key-value pairs compared to objects, especially for large collections.

```
Modifying Arrays:

push(element1, ..., elementN): Adds one or more elements to t
end of an array and returns the new length.
pop(): Removes the last element from an array and returns tha
unshift(element1, ..., elementN): Adds one or more elements t
 of an array and returns the new length.
shift(): Removes the first element from an array and returns 
splice(start, deleteCount, item1, ..., itemN): Adds/removes e
starting at the specified index.

myArray.push(6); // Adds 6 to the end of the array
console.log(myArray); // Output: [1, 2, 3, 4, 5, 6]

myArray.pop(); // Removes the last element (6) from the array
console.log(myArray); // Output: [1, 2, 3, 4, 5]

myArray.unshift(0); // Adds 0 to the beginning of the array
console.log(myArray); // Output: [0, 1, 2, 3, 4, 5]

myArray.shift(); // Removes the first element (0) from the ar
console.log(myArray); // Output: [1, 2, 3, 4, 5]

myArray.splice(2, 1); // Removes 1 element from index 2
console.log(myArray); // Output: [1, 2, 4, 5]



Accessing Elements:

concat(array1, ..., arrayN): Returns a new array combining th
on which it's called with other arrays and/or values.
slice(start, end): Extracts a section of an array and returns
```

```
indexOf(searchElement, fromIndex): Returns the first index at
element can be found in the array, or -1 if it is not present
lastIndexOf(searchElement, fromIndex): Returns the last index
given element can be found in the array, or -1 if it is not p


let newArray = myArray.concat([6, 7, 8]); // Concatenates two
console.log(newArray); // Output: [1, 2, 4, 5, 6, 7, 8]

let slicedArray = newArray.slice(1, 4); // Extracts elements
console.log(slicedArray); // Output: [2, 4, 5]

let index = newArray.indexOf(4); // Finds the index of elemen
console.log(index); // Output: 2


Iterating over Arrays:

forEach(callbackFn): Executes a provided function once for ea
map(callbackFn): Creates a new array populated with the resul
 provided function on every element in the array.
filter(callbackFn): Creates a new array with all elements tha
implemented by the provided function.
reduce(callbackFn, initialValue): Applies a function against
and each element in the array to reduce it to a single value.


newArray.forEach((element, index) => {
  console.log(`${index}: ${element}`);
});

let doubledArray = newArray.map(element => element * 2); // D
console.log(doubledArray);

let filteredArray = newArray.filter(element => element % 2 ==
console.log(filteredArray);

let sum = newArray.reduce((accumulator, currentValue) => accu
 // Calculates sum
console.log(sum);



Checking Array Content:

includes(searchElement, fromIndex): Determines whether an arr
some(callbackFn): Checks if at least one element in the array
 by the provided function.
every(callbackFn): Checks if all elements in the array pass t
provided function.

let includes = newArray.includes(3); // Checks if array inclu
console.log(includes); // Output: false

let some = newArray.some(element => element > 5); // Checks i
console.log(some); // Output: true

let every = newArray.every(element => element < 10); // Check
console.log(every); // Output: true

Sorting and Manipulating Arrays:

sort(compareFn): Sorts the elements of an array in place and
reverse(): Reverses the order of the elements in an array in
fill(value, start, end): Fills all the elements of an array f
 to an end index with a static value.


 let sortedArray = newArray.sort((a, b) => a - b); // Sorts t
console.log(sortedArray);

let reversedArray = newArray.reverse(); // Reverses the array
console.log(reversedArray);

let filledArray = newArray.fill(0, 2, 4); // Fills elements f
```

```
console.log(filledArray);
```

```
// Modifying Strings
const str1 = 'Hello';
const str2 = 'World';


Modifying Strings:
concat(str1, ..., strN): Concatenates two or more strings and
slice(start, end): Extracts a section of a string and returns
substring(start, end): Extracts characters from a string betw
indices and returns a new string.
substr(start, length): Extracts a specified number of charact
starting at the specified position, and returns a new string.
replace(searchValue, replaceValue): Searches a string for a s
regular expression, and replaces the found match with a new s
padStart(targetLength, padString): Pads the current string wi
until the resulting string reaches the given length from the
padEnd(targetLength, padString): Pads the current string with
until the resulting string reaches the given length from the
trim(): Removes whitespace from both ends of a string.
const concatenatedStr = str1.concat(' ', str2); // Output: 'H

const str = 'Hello World';
const slicedStr = str.slice(6); // Output: 'World'
const subStr = str.substring(0, 5); // Output: 'Hello'
const replacedStr = str.replace('World', 'Universe'); // Outp

const paddedStrStart = str.padStart(15, '-'); // Output: '---
const paddedStrEnd = str.padEnd(15, '-'); // Output: 'Hello W
```

```
Searching Strings:
indexOf(searchValue, fromIndex): Returns the index within the
calling string object of the first occurrence of the specifie
starting the search at fromIndex.
lastIndexOf(searchValue, fromIndex): Returns the index within
 string object of the last occurrence of the specified value,
startsWith(searchString, position): Determines whether a stri
 with the characters of a specified string.
endsWith(searchString, length): Determines whether a string e
 the characters of a specified string.
includes(searchString, position): Determines whether one stri
 be found within another string.
search(regexp): Executes a search for a match between a regul

const index = str.indexOf('World'); // Output: 6
const lastIndex = str.lastIndexOf('l'); // Output: 9

const startsWithHello = str.startsWith('Hello'); // Output: t
const endsWithWorld = str.endsWith('World'); // Output: true
const includesHello = str.includes('Hello'); // Output: true

const searchResult = str.search(/W/); // Output: 6


Transforming Strings:
toUpperCase(): Converts a string to uppercase letters.
toLowerCase(): Converts a string to lowercase letters.
toLocaleUpperCase(locale): Returns a new string whose textual
are the same as this string but whose uppercase letters are c
locale-specific uppercase letters.
toLocaleLowerCase(locale): Returns a new string whose textual
are the same as this string but whose lowercase letters are c
locale-specific lowercase letters.

const upperCaseStr = str.toUpperCase(); // Output: 'HELLO WOR
const lowerCaseStr = str.toLowerCase(); // Output: 'hello wor
const localUpperStr = str.toLocaleUpperCase('en-US'); // Outp
const localLowerStr = str.toLocaleLowerCase('en-US'); // Outp

Splitting and Joining Strings:
split(separator, limit): Splits a string into an array of sub
specified separator and returns a new array.
join(separator): Joins all elements of an array into a string

const splitStr = str.split(' '); // Output: ['Hello', 'World'
const joinedStr = splitStr.join('-'); // Output: 'Hello-World


Extracting and Checking String Content:
charAt(index): Returns the character at the specified index.
charCodeAt(index): Returns the Unicode of the character at th
codePointAt(pos): Returns a non-negative integer that is the
substring(start, end): Extracts characters from a string betw
indices and returns a new string.
slice(start, end): Extracts a section of a string and returns
trim(): Removes whitespace from both ends of a string.

const charAtIndex = str.charAt(6); // Output: 'W'
const charCodeAtIndex = str.charCodeAt(6); // Output: 87
const codePointAtPos = str.codePointAt(6); // Output: 87
const trimmedStr = str.trim(); // Output: 'Hello World'

Encoding and Decoding:
encodeURIComponent(str): Encodes a URI component.
decodeURIComponent(str): Decodes a URI component.
encodeURI(str): Encodes a URI.
decodeURI(str): Decodes a URI.

const encodedURI = encodeURIComponent('Hello World'); // Outp
const decodedURI = decodeURIComponent('Hello%20World'); // Ou
const encodedURIComponent = encodeURI('Hello World'); // Outp
const decodedURIComponent = decodeURI('Hello%20World'); // Ou

Miscellaneous:
```

```
repeat(count): Returns a new string with a specified number o
string it was called on.
localeCompare(compareString): Compares two strings in the cur
match(regexp): Used to match a regular expression against a s
toLocaleLowerCase() and toLocaleUpperCase(): Returns a new st
locale-specific case transformation applied.
const repeatedStr = str.repeat(2); // Output: 'Hello WorldHel
const localeComparison = 'ä'.localeCompare('z'); // Output: -
const matchedStr = str.match(/Hello/); // Output: ['Hello', i
```

Shallow copy and deep copy are two ways to duplicate objects in JavaScript, each with its own characteristics:

1. **Shallow Copy:**
    - A shallow copy creates a new object and then copies the values of the original object's properties into the new object.
    - If the original object contains nested objects (objects within objects), only the references to those nested objects are copied, not the nested objects themselves. Thus, changes made to nested objects in the copied object will affect the original object and vice versa.
    - Shallow copy can be created using methods like `Object.assign()`, spread operator (`...`), or `Array.slice()` for arrays.

```
// Using Object.assign()
const originalObject = { a: 1, b: { c: 2 } };
const shallowCopy = Object.assign({}, originalObject);

// Using spread operator
const shallowCopy = { ...originalObject };

// For arrays
```

```
const originalArray = [1, 2, [3, 4]];
const shallowCopyArray = originalArray.slice();
```

1. **Deep Copy:**
    - A deep copy creates a new object and then recursively copies all properties of the original object, including nested objects, into the new object. This means that even nested objects are duplicated, and changes made to them in the copied object will not affect the original object and vice versa.
    - Deep copy is more complex and requires custom implementations, such as recursive functions or libraries like Lodash.

```
// Using JSON.parse() and JSON.stringify()
const originalObject = { a: 1, b: { c: 2 } };
const deepCopy = JSON.parse(JSON.stringify(originalObject));

// Custom recursive function (not handling circular reference
function deepCopy(obj) {
  if (typeof obj !== 'object' || obj === null) {
    return obj;
  }

  let newObj = Array.isArray(obj) ? [] : {};

  for (let key in obj) {
    newObj[key] = deepCopy(obj[key]);
  }

  return newObj;
}

const deepCopy = deepCopy(originalObject);

// Using external libraries like Lodash
const deepCopy = _.cloneDeep(originalObject);
```

Scope

- Scope in JavaScript determines the accessibility of variables and functions at various parts of one's code or program.
- In other words, Scope will help us to determine a given part of a code or a program, what variables or functions one can access, and what variables or functions one cannot access.
- Within a scope itself, a variable a function, or a method could be accessed. Outside the specified scope of a variable or function, the data cannot be accessed.
- There are three types of scopes available in JavaScript: **Global Scope, Local / Function Scope,** and **Block Scope.**

Global Scope:

- Variables or functions (or methods) that are declared under a global namespace (like area or location) are determined as Global Scope.
- It means that all the variables that have global scope can be easily accessed from anywhere within the code or a program.

```
// Global Scoped letiable
let global_letiable = "GeeksforGeeks";

// First function...
function first_function() {
    return global_letiable;
}

// Second function...
function second_function() {
    return first_function();
}
```

```
console.log(second_function());
```

Local or Function Scope:

- Variables that are declared inside a function or a method have Local or Function Scope.
- It means those variables or functions which are declared inside the function or a method can be accessed within that function only.

```
function main_function() {

    // letiable with Local Scope...
    let a = 2;

    // Nested Function having Function Scope
    let multiply = function () {

        // It can be accessed and altered as well
        console.log(a * 5);
    }

    // Will be called out when main_function gets called
    multiply();
}

// Display's the result...
console.log(main_function());

// Throws a reference error since it
// is a locally scoped letiable
console.log(a);

// Throws a reference error since it
// is a locally scoped function
multiplyBy2();
```

Block Scope:

- **Block Scope** is related to the variables or functions which are declared using **let** and **const** since **var** does not have block scope.
- Block Scope tells us that variables that are declared inside a block { }, can be accessed within that block only, not outside of that block.

```
{
    let x = 13;
}
// Throws a reference error
// since x is declared inside a block which
// cannot be used outside the block
console.log(x);
```

Scope Chain

- JavaScript engine uses scopes to find out the exact location or accessibility of variables and that particular process is known as Scope Chain.
- Scope Chain means that one variable has a scope (it may be global or local/function or block scope) is used by another variable or function having another scope (may be global or local/function or block scope).

```
function outer() {
    const outerVar = 'Outer Variable';
    function inner() {
        console.log(outerVar); // Inner function can access
outerVar
    }
    inner();
}

outer(); // Output: Outer Variable
```

Lexical scope

The scope is determined by the placement of variables and functions in the code, and it remains the same throughout the execution of the program. **Global variables** can be accessed from anywhere within the program, while local variables can only be accessed within the function or block in which they are defined. The nested scope allows functions to access variables defined in parent functions, and block scope allows variables to have limited accessibility within a block of code.

lexical scope vs clousres

Lexical Scope:

- Lexical scope, also known as static scope, is a concept that determines the accessibility of variables based on their location in the source code during the lexical analysis phase of compilation.
- In lexical scoping, the scope of a variable is defined by its location within the code's nested block structure at the time of authoring, rather than at runtime.
- Variables are resolved by looking up their references in the hierarchy of nested scopes as determined by the code's structure.
- Lexical scoping dictates where a variable is accessible within a program's codebase.

```
function outer() {
    const outerVar = 'Outer Variable';
    function inner() {
        console.log(outerVar); // Inner function can access
outerVar
    }
    inner();
}
outer(); // Output: Outer Variable
```

Closures:

- Closures are a feature in programming languages, including JavaScript, that allow functions to retain access to variables from their lexical scope even after the enclosing function has finished executing.
- A closure is formed when an inner function is defined within an outer function and references variables from the outer function's scope.
- Closures "close over" or capture the environment (lexical scope) in which they are defined, preserving access to variables even after the outer function has returned.
- Closures enable functions to maintain and access private state, providing a means of encapsulating and preserving data.

=

```
function outer() {
    const outerVar = 'Outer Variable';
    return function inner() {
        console.log(outerVar); // Inner function accesses o
uterVar via closure
    };
}
const closureFunction = outer();
closureFunction(); // Output: Outer Variable
```

Differences:

- Lexical scope defines where variables are accessible based on their location in the source code during authoring.
- Closures enable functions to retain access to variables from their enclosing lexical scope, even after the enclosing function has returned.
- Lexical scope is a static property determined at compile time, while closures are dynamic runtime constructs created when functions are defined and executed.

Higher-order functions

Higher-order functions are functions that can take one or more functions as arguments or return functions as their results.

One of several benefits of higher-order functions is that they enable code reusability. Instead of writing similar logic, such as filtering, multiple times, you can encapsulate it in a higher-order function once and pass different functions as arguments to achieve different behaviors.

```
// Using the iterative approach (imperative).

const numbers = [1, 2, 3, 4, 5, 6, 7, 8];

const evenNumbers = [];

for (let i = 0; i < numbers.length; i++) {
    if (number[i] % 2 == 0) {
        evenNumbers.push(numbers[i]);
    }
}

console.log(evenNumbers); // [2, 4, 6, 8]

// Using a higher-order function (declarative).

const numbers = [1, 2, 3, 4, 5, 6, 7, 8];

const evenNumbers = numbers.filter(number => number % 2 === 0
```

Map:

- The `map` function applies a callback function to each element of an array and returns a new array containing the results of applying the callback function to each element.
- It does not mutate the original array; instead, it creates and returns a new array with the transformed elements.

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map(num => num * 2);
// doubled is [2, 4, 6, 8, 10]
```

Filter:

- The `filter` function creates a new array with all elements that pass the test implemented by the provided callback function.
- It does not mutate the original array; it returns a new array containing only the elements for which the callback function returns `true`.

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

Reduce:

- The `reduce` function applies a callback function to each element of an array, resulting in a single output value.
- It can be used to perform operations like summing all elements, calculating the product of all elements, or any other operation that reduces the array to a single value.
- The callback function takes four arguments: accumulator, currentValue, currentIndex, and the array itself. The accumulator is the accumulated result of previous iterations.

```
const numbers = [1, 2, 3, 4, 5];
const sum = numbers.reduce((accumulator, currentValue) => acc
// sum is 15
```

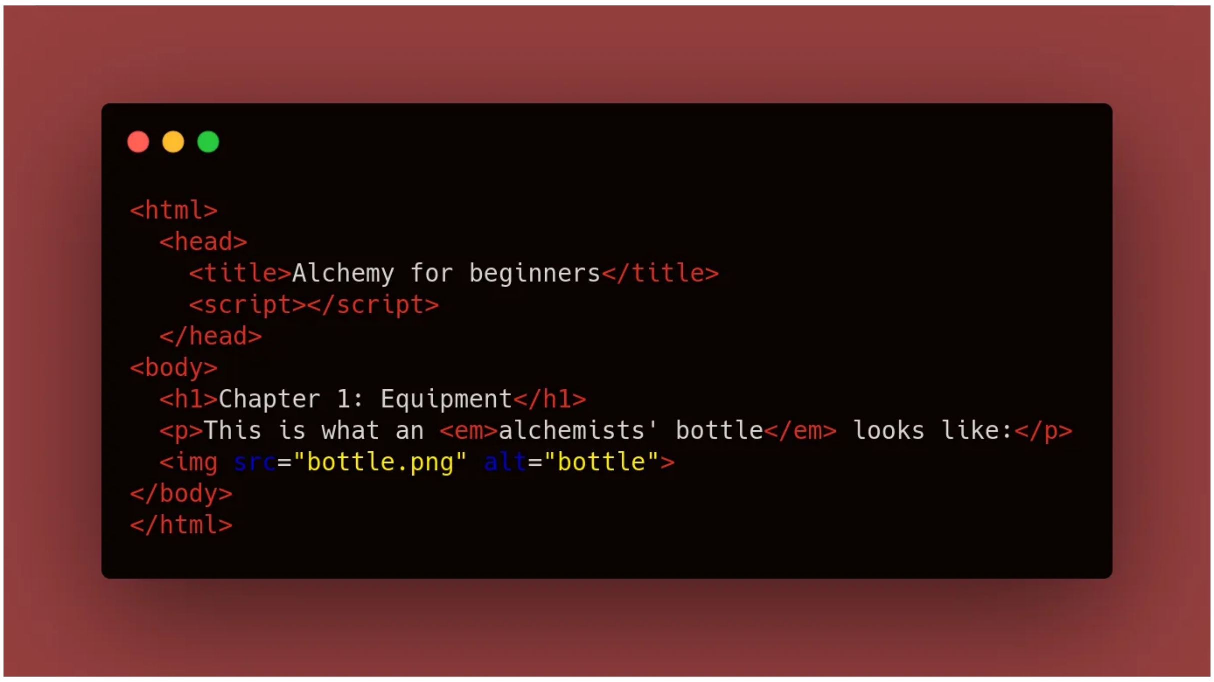
what is primary distinction between array.foreach() vs map()

1. `Array.forEach()`:

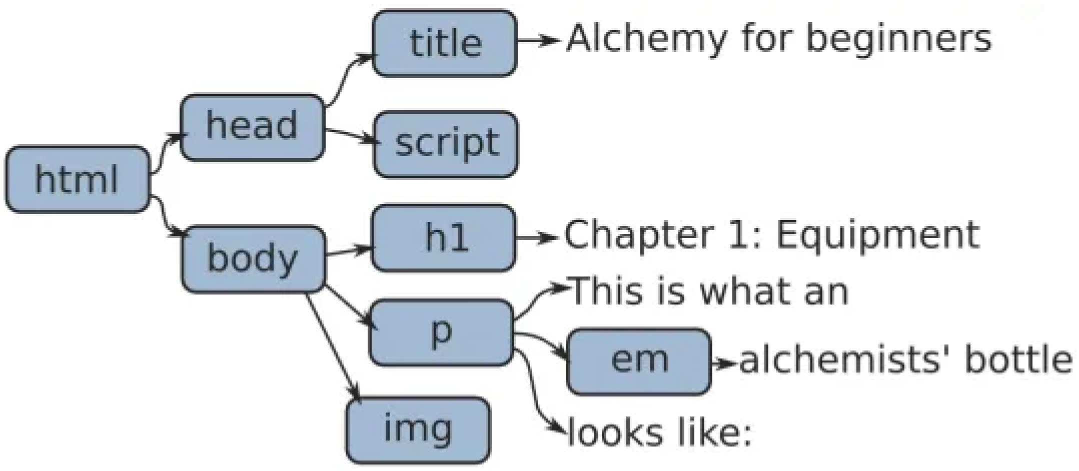
- `Array.forEach()` is a method used to iterate over each element in an array and execute a provided callback function for each element.
- It doesn't return anything (or returns `undefined` explicitly).
- The purpose of `forEach()` is to perform an action or side effect for each element in the array, such as logging values, updating variables, or performing operations that don't necessarily result in a new array.

2. `Array.map()`:
    - `Array.map()` is a method used to iterate over each element in an array, apply a provided callback function to each element, and return a new array containing the results of applying the callback function to each element.
    - It returns a new array with the same length as the original array, where each element is the result of applying the callback function to the corresponding element in the original array.
    - The purpose of `map()` is to transform each element in the array in a predictable and consistent way, creating a new array with the transformed elements.

a website consists of an HTML and CSS document. The browser creates a representation of that document known as Document Object Model (DOM).
This document enables Javascript to **access** and **manipulate** the elements and styles of a website and also add functionality to our page.
The model is built in a tree structure of objects. Everything which is an element in HTML is an object in DOM.



```html
<html>
  <head>
    <title>Alchemy for beginners</title>
    <script></script>
  </head>
<body>
  <h1>Chapter 1: Equipment</h1>
  <p>This is what an <em>alchemists' bottle</em> looks like:</p>
  <img src="bottle.png" alt="bottle">
</body>
</html>
```

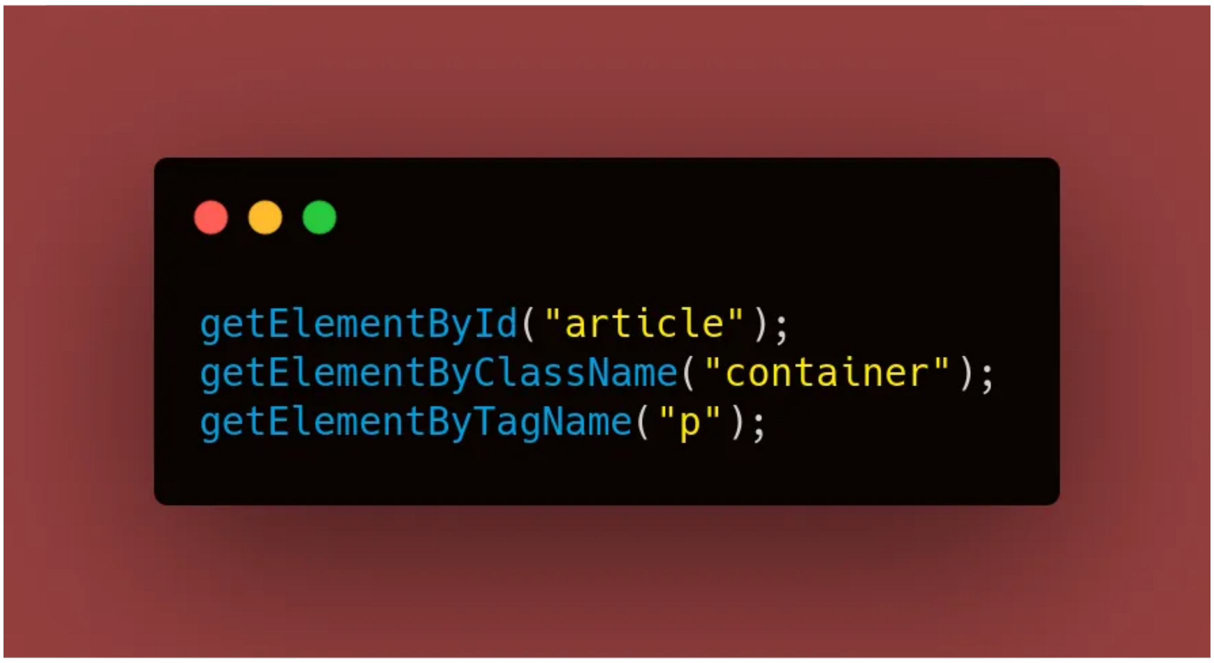


Accessing the elements

To apply styles or functionality to an element using JS, we need to first select that element and then do what we have to do to that element. For accessing, DOM provides many inbuilt functions that we can use to select that element.

getElementBy

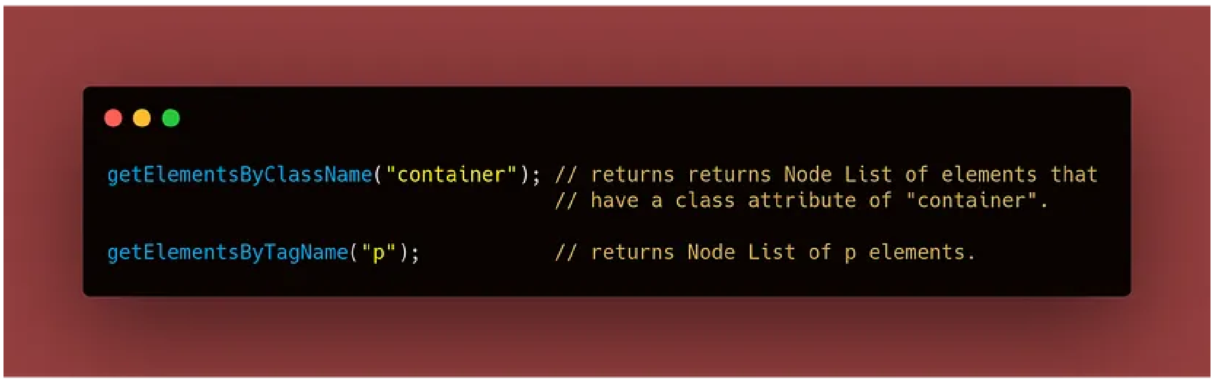
The **getElementBy[Id|ClassName|TagName]()** method returns the element that has the ID attribute, Class attribute or Tag attribute with the specified value. For eg,



```
getElementById("article");
getElementByClassName("container");
getElementByTagName("p");
```

getElementsBy

The **getElementsBy[ClassName|TagName]()** method returns all the elements that have the Class attribute or Tag attribute with the specified value in node form. For eg,

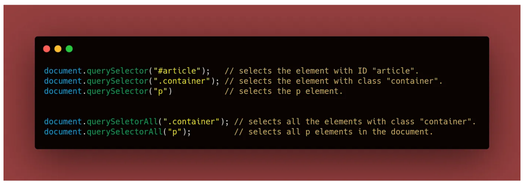


```
getElementsByClassName("container"); // returns returns Node List of elements that
                                     // have a class attribute of "container".

getElementsByTagName("p");           // returns Node List of p elements.
```

querySelector & querySelectorAll

document.querySelector() returns the first element within the document that matches the specified selector, or group of selectors. If no matches are found, null is returned. We don't have to use different methods for different tags.

document.querySelectorAll() returns a Node List representing a list of the document's elements that match the specified group of selectors. If no matches are found, null is returned.



```
document.querySelector("#article");   // selects the element with ID "article".
document.querySelector(".container"); // selects the element with class "container".
document.querySelector("p")           // selects the p element.


document.querySeletorAll(".container"); // selects all the elements with class "container".
document.querySelectorAll("p");         // selects all p elements in the document.
```

What is the purpose of the addEventListener method?

`addEventListener` is a built-in function in JavaScript that allows you to attach an event handler to a specified element, such as a button or a link.
The `addEventListener` function takes two arguments: the type of the event you want to listen for (e.g. "click" or "keydown") and the function that should be called when the event is detected.

```
let button = document.querySelector('#my-button');
button.addEventListener('click', function() {
  console.log('Button was clicked');
});
```

In this example, we're selecting the button with an ID of "my-button" and adding a `click` event listener to it. When the button is clicked, the anonymous function will be called, and "Button was clicked" will be logged to the console.

. How do you create and remove elements in the DOM?

Creating Elements:

1. **createElement**: You can use the `document.createElement()` method to create a new HTML element. For example:

```
var newElement = document.createElement('div');
```

appendChild or insertBefore: Once you have created an element, you can insert it into the DOM using either `appendChild()` or `insertBefore()` method. For exampl

```
var parentElement = document.getElementById('parent');
parentElement.appendChild(newElement);
// or
var siblingElement = document.getElementById('sibling');
parentElement.insertBefore(newElement, siblingElement);
```

Removing Elements:

1. **removeChild**: To remove an element from the DOM, you can use the `removeChild()` method. You need to call this method on the parent element of the element you want to remove. For example:

```
var parentElement = document.getElementById('parent');
var childElement = document.getElementById('child');
parentElement.removeChild(childElement);
```

1.

remove: Alternatively, you can directly call the `remove()` method on the element you want to remove. This method removes the element from the DOM. For example:

```
var elementToRemove = document.getElementById('element');
elementToRemove.remove();
```

What is event propagation and Event Delegation

Event propagation and event delegation are important concepts in JavaScript that help manage event handling efficiently, especially in scenarios where multiple elements are involved.

Event Propagation:

Event propagation refers to the way events traverse through the DOM tree. When an event occurs on an element in the DOM, such as a click event, the event can propagate in two phases:

1. **Capturing Phase (`capture`)**: In this phase, the event starts from the outermost ancestor element and moves towards the target element. This phase allows you to capture the event before it reaches the target element.
2. **Bubbling Phase (`bubble`)**: In this phase, the event starts from the target element and moves up the DOM tree towards the outermost ancestor element. This phase allows you to handle the event as it bubbles up through the DOM tree.

Event propagation can be influenced by event handlers and the `addEventListener()` method, which allows you to specify whether to handle events during the capturing or bubbling phase.

Event Delegation:

Event delegation is a technique that leverages event propagation to efficiently manage event handling for multiple elements. Instead of attaching event handlers to individual elements, you attach a single event handler to a common ancestor element that exists when the page is initially loaded.

When an event occurs, such as a click, it bubbles up through the DOM tree. The event handler attached to the common ancestor intercepts the event during the bubbling phase. By examining the event target (the element where the event originated), you can determine which specific child element triggered the event. This allows you to handle events for dynamically added elements or a large number of elements more efficiently.

```html
<ul id="parent-list">
  <li>Item 1</li>
  <li>Item 2</li>
  <li>Item 3</li>
</ul>
```

```javascript
// Attach event listener to the parent list
document.getElementById('parent-list').addEventListener('clic
  // Check if the clicked element is an <li> element
  if (event.target.tagName === 'LI') {
    // Perform action based on the clicked <li> element
    console.log('Clicked on:', event.target.textContent);
  }
});
```

In this example, instead of attaching a click event listener to each `<li>` element individually, we attach a single event listener to the parent `<ul>` element. The event handler checks if the clicked element is an `<li>` element and performs the appropriate action. This approach is more efficient, especially for large lists or dynamically added elements, as it reduces the number of event listeners needed.

what is set timeout and set interval

setTimeout:

The `setTimeout` function executes a given piece of code or a function after a specified delay (in milliseconds).

```javascript
setTimeout(function() {
  console.log("Delayed function executed!");
}, 2000); // Executes after 2 seconds
```

setInterval:

The `setInterval` function repeatedly executes a given piece of code or a function at specified intervals (in milliseconds).

```
setInterval(function() {
  console.log("Repeatedly executed function!");
}, 1000); // Executes every 1 second
```

How They Work:

1. **setTimeout**: When you call `setTimeout`, JavaScript sets up a timer to delay execution of the function by the specified amount of time. Once the delay is over, the function is moved to the callback queue, and when the call stack is empty, the function is executed.
2. **setInterval**: `setInterval` works similarly to `setTimeout`, but instead of executing the function once after a delay, it repeatedly schedules the execution of the function at specified intervals. The function continues to execute at the specified interval until `clearInterval` is called or the page is unloaded.

What is the purpose of the arguments object in a function?

The `arguments` object in JavaScript is a local variable available within all functions. It allows access to all the arguments passed to a function, regardless of the number of parameters defined in the function signature. The purpose of the `arguments` object is to provide flexibility and versatility when working with functions, especially when the number of arguments passed to a function is unknown or variable.

Accessing Function Arguments: You can access individual arguments passed to a function using numerical indices within the `arguments` object. This is particularly useful when the number of arguments is not known in advance or when the function is designed to accept a variable number of arguments

```
function sum() {
  var total = 0;
  for (var i = 0; i < arguments.length; i++) {
    total += arguments[i];
  }
  return total;
}

console.log(sum(1, 2, 3)); // Output: 6
console.log(sum(10, 20, 30, 40)); // Output: 100
```

Creating Functions with Variable Arguments: The `arguments` object enables you to create functions that accept a variable number of arguments without explicitly defining parameters in the function signature

```
function concatenate() {
  var result = '';
  for (var i = 0; i < arguments.length; i++) {
    result += arguments[i];
  }
  return result;
}

console.log(concatenate('Hello', ' ', 'world')); // Output: H
console.log(concatenate('JavaScript', ' ', 'is', ' ', 'awesom
```

Arrow functions, Template literals, Destructuring spread and rest

Arrow Functions:

Arrow functions provide a concise syntax for writing functions in JavaScript. They have a more compact syntax compared to traditional function expressions and automatically capture the `this` value of the enclosing context.

```
// Traditional function expression
function add(a, b) {
    return a + b;
}

// Arrow function
const add = (a, b) => a + b;
```

Template Literals:

Template literals allow for more flexible and readable string formatting by enabling string interpolation and multiline strings. They use backticks (`) instead of single or double quotes.

```
const name = "Alice";
const greeting = `Hello, ${name}!`;
console.log(greeting); // Outputs: "Hello, Alice!"
```

Destructuring:

Destructuring enables extracting values from arrays or properties from objects into distinct variables, making it easier to work with complex data structures.

Array Destructuring:

```
const numbers = [1, 2, 3];
const [first, second] = numbers;
console.log(first, second); // Outputs: 1 2
```

Object Destructuring:

```
const person = { name: "Alice", age: 30 };
const { name, age } = person;
console.log(name, age); // Outputs: "Alice" 30
```

Spread Syntax:

Spread syntax (`...`) allows an iterable (like an array) to be expanded into individual elements. It can be used in function calls, array literals, and object literals to copy properties.

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

Rest Parameters:

Rest parameters allow a function to accept an indefinite number of arguments as an array, making it easier to work with variable-length argument lists.

```


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

console.log(sum(1, 2, 3, 4)); // Outputs: 10
```

What is a common scenario for employing anonymous functions?

Anonymous functions, also known as function expressions, are often employed in scenarios where a function is needed as a parameter to another function, or when you want to define a function locally without giving it a name.

Callback Functions: Anonymous functions are frequently used as callback functions, which are passed as arguments to other functions and executed asynchronously or in response to events.

```
// Example: Array.prototype.map()
const numbers = [1, 2, 3];
const squaredNumbers = numbers.map(function(x) {
    return x * x;
});
console.log(squaredNumbers); // Outputs: [1, 4, 9]
```

Event Handlers: When attaching event handlers to HTML elements, you often use anonymous functions to define the behavior triggered by the event.

```
// Example: Adding a click event handler
const button = document.querySelector('button');
button.addEventListener('click', function() {
    console.log('Button clicked!');
});
```

Immediately Invoked Function Expressions (IIFEs): Anonymous functions can be invoked immediately after their definition, creating a private scope for encapsulating variables and avoiding global namespace pollution.

```
// Example: IIFE for creating a private scope
(function() {
    var privateVariable = 'This is private';
    console.log(privateVariable);
})();
```

Asynchronous Operations: In asynchronous programming, anonymous functions are often used as callbacks for handling the result of asynchronous operations like fetching data or performing actions after a delay.

```
// Example: setTimeout with an anonymous function
setTimeout(function() {
    console.log('Delayed operation');
}, 1000);
```

List the advantages and disadvantages of extending built-in JavaScript objects.

Extending built-in JavaScript objects, such as Array, String, Object, etc., means adding custom methods or properties to their prototypes.

Advantages of extending built-in objects:

1. Enhanced code reusability.
2. Improved convenience and readability.
3. Seamless integration with existing JavaScript functionalities.
4. Facilitation of polyfilling missing features.

Disadvantages of extending built-in objects:

1. Global scope pollution.
2. Increased risk of name collisions.
3. Maintenance challenges.
4. Compatibility concerns, particularly with legacy code or older browsers.
5. Potential introduction of security vulnerabilities.

Define what a promise is and describe its applications.

A promise is an object representing the eventual completion or failure of an asynchronous operation. It allows you to handle asynchronous operations more easily by providing a cleaner and more intuitive way to work with asynchronous code.

Here's a breakdown of key concepts related to promises:

1. **State**: A promise can be in one of three states: pending, fulfilled, or rejected. When a promise is created, it's initially in a pending state. It transitions to either a fulfilled state (meaning the operation was successful) or a rejected state (meaning the operation failed).
2. **Handlers**: Promises provide methods such as `then()` and `catch()` to handle the result of an asynchronous operation. The `then()` method is used to handle the fulfillment of the promise, while the `catch()` method is used to handle any errors that occur during the execution of the asynchronous operation.

3. **Chaining**: Promises allow you to chain multiple asynchronous operations together using the `then()` method. This enables you to execute a series of asynchronous tasks sequentially, making the code more readable and easier to manage.
4. **Error Handling**: Promises provide a standardized way to handle errors in asynchronous code. Any errors that occur during the execution of a promise are automatically propagated down the promise chain until they are caught by a `catch()` handler.

Can you provide an illustration of how ES6 has altered the approach to working with "this" in JavaScript?

Arrow Functions:

In ES6, arrow functions (`=>`) were introduced, which have a different behavior regarding the `this` keyword compared to traditional function expressions. Arrow functions do not have their own `this` context; instead, they inherit the `this` value from the enclosing lexical scope. This behavior makes arrow functions particularly useful for avoiding the confusion of `this` binding in nested functions or callback functions.

```javascript
// Traditional function expression
function greet() {
    console.log("Hello, " + this.name);
}

const person = {
    name: "Alice",
    greet: greet
};

person.greet(); // Outputs: "Hello, Alice"

// Arrow function
const greetArrow = () => {
    console.log("Hello, " + this.name);
};

const personArrow = {
    name: "Alice",
    greet: greetArrow
};

personArrow.greet(); // Outputs: "Hello, undefined"
```

Classes:

ES6 introduced class syntax for defining constructor functions and prototypes in a more structured and intuitive way. Inside class methods, `this` behaves similarly to traditional functions, referring to the instance of the class that calls the method.

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

    greet() {
        console.log("Hello, " + this.name);
    }
}

const person = new Person("Alice");
person.greet(); // Outputs: "Hello, Alice"
```

Distinguish between host objects and native objects.

- Host objects are provided by the JavaScript environment (e.g., web browser or Node.js), while native objects are built-in to the JavaScript language itself.
- Host objects' behavior may vary between environments, while native objects have standardized behavior across all JavaScript environments.

- Host objects are not defined by the ECMAScript specification and may have non-standard features, while native objects are defined by the ECMAScript specification and provide core functionality.
- Examples of host objects include `window` and `document` in a web browser, while examples of native objects include `Array`, `String`, `Object`, and `Math`.

What is type coercion, and what are some common pitfalls associated with relying on it in JavaScript code?

Type coercion is the process of converting the type of a value to another type during runtime in JavaScript. JavaScript is a dynamically typed language, which means that variables can hold values of any type, and type coercion happens implicitly when performing operations involving values of different types.

Common examples of type coercion in JavaScript include:

- Conversion of a non-Boolean value to a Boolean in contexts that require a Boolean (e.g., in an `if` statement).
- Conversion of values to strings when using the addition operator (`+`).

While type coercion can sometimes be convenient, it can also lead to unexpected behavior and bugs in JavaScript code.

- **Loss of Precision**: Coercion may result in loss of precision, especially in numeric conversions.
- **Implicit Conversion**: Coercion obscures code intent, making it less readable and maintainable, particularly for unfamiliar developers.
- **Unexpected Truthy/Falsy Values**: Coercion can yield unexpected truthy or falsy values, especially in Boolean contexts with non-Boolean values.
- **Debugging Complexity**: Coercion complicates debugging by masking reasons for type conversions, adding to code complexity.

To avoid common pitfalls associated with type coercion, it's essential to:

- Be aware of JavaScript's coercion rules and understand how different types are coerced in various contexts.

- Use explicit type conversion methods (`parseInt()`, `parseFloat()`, `toString()`, `Number()`, `Boolean()`, etc.) when necessary to make the code clearer and more predictable.
- Use strict comparison operators (`===` and `!==`) instead of loose equality operators (`==` and `!=`) to avoid unexpected coercion.

Describe the advantages of using the arrow syntax for methods in constructors.

Using the arrow syntax for methods in constructors, or any JavaScript object, offers several advantages:

1. **Lexical `this` Binding**: Arrow functions do not have their own `this` context; instead, they inherit the `this` value from the enclosing lexical scope. In the context of constructors, this means that arrow functions defined within constructors will always retain the `this` value of the surrounding constructor, eliminating the need to bind `this.`
2. **Simplified Syntax**: Arrow functions have a concise syntax, lead to cleaner and more readable code, reducing the need for boilerplate code associated with traditional function expressions.
3. **Implicit Return**: Arrow functions automatically return the result of the expression without needing an explicit `return` statement if the function body consists of a single expression.
4. **Avoidance of Function Binding**: Traditional function expressions create a new `this` context when called as methods of an object. Arrow functions, on the other hand, do not create their own `this` context, making them suitable for use as methods within constructors without needing to bind `this`.
5. **Improved Compatibility with Event Handlers and Callbacks**: Since arrow functions inherit the `this` value from their surrounding lexical scope, they are particularly useful for event handlers and callbacks where maintaining the correct `this` context can be crucial. This simplifies code and reduces the need for workarounds like using `bind()` or creating closures.

Discuss the same-origin policy's implications for JavaScript.

The same-origin policy is a crucial security measure implemented by web browsers to prevent malicious scripts from accessing sensitive data across different origins (domains, protocols, or ports). In JavaScript, this policy has significant implications, primarily revolving around security and data integrity.

1. **Cross-Origin Requests**: JavaScript running in a web page is typically subject to the same-origin policy, which restricts the ability to make cross-origin requests via XMLHttpRequest or fetch API. This means that scripts on one origin cannot directly make requests to resources on another origin unless explicitly permitted by Cross-Origin Resource Sharing (CORS) headers.
2. **Security**: The same-origin policy helps prevent malicious scripts from accessing sensitive data, such as cookies, session tokens, or user information, from other origins.
3. **Data Isolation**: JavaScript code executing in the context of one origin cannot directly access or modify the Document Object Model (DOM) of a document from a different origin.
4. **Script Inclusion**: JavaScript files loaded from different origins are also subject to the same-origin policy. This means that a script included from a different origin (e.g., via a script tag or dynamically injected script) is restricted in its access to resources and interaction with the DOM of the hosting page.
5. **Cross-Origin Communication**: While direct cross-origin communication is restricted by the same-origin policy, there are mechanisms available to facilitate communication between scripts from different origins, such as Cross-Origin Messaging (postMessage) and Cross-Origin Resource Sharing (CORS). These mechanisms allow controlled data exchange between cooperating origins while maintaining security.

Define strict mode and outline some of its advantages and disadvantages

Strict mode is a feature introduced in ECMAScript 5 (ES5) that allows developers to opt into a stricter set of rules and behaviors for JavaScript code execution. When strict mode is enabled, the JavaScript engine enforces additional constraints and throws more errors to help developers write cleaner, more robust code.

Advantages:

1. **Error Detection**: Strict mode helps catch common programming errors by enabling stricter parsing and error handling.
2. **Elimination of Silent Errors**: In non-strict mode, some JavaScript behaviors are allowed that can lead to unexpected results, such as creating global variables unintentionally or using reserved keywords. Strict mode prevents these silent errors, promoting cleaner and more predictable code.
3. **Security Improvements**: Strict mode mitigates certain security vulnerabilities by prohibiting dangerous or insecure features
4. **Performance Optimizations**: Strict mode can enable optimizations by the JavaScript engine, as it restricts certain language features that might otherwise have performance overhead. This can lead to faster execution of code in some cases.

Disadvantages:

1. **Backward Compatibility**: Strict mode introduces breaking changes to existing code that relies on non-strict behavior. Enabling strict mode in an existing codebase may require significant refactoring and testing to ensure compatibility with older browsers or environments.
2. **Learning Curve**: Developers who are not familiar with strict mode may find it challenging to understand its nuances and implications.
3. **Compatibility Issues**: While most modern browsers and JavaScript environments support strict mode, there may still be compatibility issues with older browsers or legacy codebases.
4. **Less Flexible**: Strict mode restricts certain language features and behaviors, which can limit flexibility in certain scenarios.

Writing JavaScript code using a language that compiles to JavaScript, often referred to as a transpiled language, has become increasingly popular due to the benefits it offers.

Pros:

1. **Improved Developer Experience**: Transpiled languages often offer modern language features, syntax enhancements, and tooling that can significantly

improve the developer experience. This includes features like static typing, pattern matching, and enhanced error-checking, which can lead to more efficient development workflows and fewer bugs.

2. **Cross-Platform Compatibility**: By compiling to JavaScript, transpiled languages enable developers to write code that can run on any platform or browser that supports JavaScript.
3. **Performance Optimization**: Some transpiled languages offer performance optimizations that can lead to faster execution compared to handwritten JavaScript. This includes features like tree shaking, dead code elimination, and advanced optimization techniques that can result in smaller bundle sizes and improved runtime performance.
4. **Language Features**: Transpiled languages often introduce language features and syntactic sugar that may not be available in vanilla JavaScript. This can include features inspired by other programming languages, such as async/await syntax for asynchronous programming
5. **Community and Ecosystem**: Many transpiled languages have vibrant communities and ecosystems with libraries, frameworks, and tools that can streamline development tasks.

Cons:

1. **Learning Curve**: Adopting a transpiled language often requires learning new syntax, language constructs, and tooling, which can pose a learning curve for developers who are unfamiliar with the language. This initial investment in learning may slow down development initially.
2. **Tooling Overhead**: Transpiled languages often require additional tooling, such as compilers, build systems, and development environments, to translate code into JavaScript. Managing this tooling and integrating it into existing workflows can add complexity and overhead to the development process.
3. **Debugging and Tool Support**: Debugging transpiled code can be more challenging compared to debugging handwritten JavaScript, especially when source maps are not properly configured.
4. **Performance Overhead**: While transpiled languages may offer performance optimizations, the compilation process itself can introduce overhead, especially for larger codebases. This can lead to longer build times and slower iteration cycles during development.

5. **Dependency on Compiler**: Using a transpiled language introduces a dependency on the compiler and build tools, which may become a single point of failure or maintenance burden.

In summary, writing JavaScript code in a language that compiles to JavaScript offers many benefits in terms of developer experience, cross-platform compatibility, and performance optimization. However, it also comes with challenges such as a learning curve, tooling overhead, and potential performance overhead.

What tools and techniques do you employ for debugging JavaScript code?

Debugging JavaScript code effectively requires a combination of tools and techniques tailored to different scenarios and environments. Here are some commonly used tools and techniques for debugging JavaScript:

1. **Browser Developer Tools**: Most modern web browsers come with built-in developer tools that offer robust debugging capabilities. These tools typically include features like breakpoints, step-by-step execution, variable inspection, and real-time code editing.
2. **Console Logging**: Console logging is a simple yet effective technique for debugging JavaScript code.
3.
4. **Debugger Statement**: The debugger statement is a built-in JavaScript keyword that triggers a breakpoint in the code when encountered.
5. **Source Maps**: Source maps provide a mapping between minified or transpiled JavaScript code and its original source code, making it easier to debug optimized or generated code. When source maps are enabled in browser developer tools, developers can debug code written in languages like TypeScript, CoffeeScript, or JSX as if it were written in plain JavaScript.
6. **Error Handling**: Proper error handling techniques, such as try-catch blocks and error logging, can help identify and diagnose runtime errors in JavaScript code. By catching and logging errors with relevant context information, developers can quickly pinpoint the root cause of issues and implement appropriate fixes.
7. **Code Linting**: Code linting tools like ESLint or JSHint analyze JavaScript code for potential errors, stylistic inconsistencies, and best practices

violations. By running code linters as part of the development workflow, developers can catch common mistakes and enforce coding standards, reducing the likelihood of bugs and improving code quality.

8. **Remote Debugging**: In scenarios where JavaScript code is running on remote devices or environments, remote debugging tools can be invaluable. These tools allow developers to connect to remote browsers or devices and debug JavaScript code remotely using browser developer tools or dedicated remote debugging clients.

Compare ES6 classes and ES5 function constructors, and furnish a use case for the arrow (⇒) function syntax.

ES6 Classes:

1. **Syntax**: ES6 classes introduce a more concise syntax for defining constructor functions and managing prototypes, resembling traditional class-based languages like Java or C++. They use the `class` keyword to declare a class and `constructor` keyword to define the constructor method.
2. **Inheritance**: ES6 classes support inheritance through the `extends` keyword, allowing classes to inherit properties and methods from a parent class using prototype chaining.
3. **Encapsulation**: ES6 classes provide better support for encapsulation, as class methods and properties can be declared as public, private, or protected using access modifiers.

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

  greet() {
    return `Hello, my name is ${this.name} and I am ${this.ag
  }
}
```

ES5 Function Constructors:

1. **Syntax**: ES5 function constructors involve defining constructor functions using regular functions and manually setting prototype methods and properties. They use the `function` keyword to define the constructor function and attach methods to the prototype manually.
2. **Inheritance**: ES5 function constructors can achieve inheritance through prototype chaining by manually setting the prototype of child constructor functions to the prototype of parent constructor functions.
3. **Encapsulation**: ES5 function constructors provide limited support for encapsulation, as JavaScript does not have native support for access modifiers. However, closures can be used to emulate private variables and methods.

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

Person.prototype.greet = function() {
  return `Hello, my name is ${this.name} and I am ${this.age}
};
```

Pass by Value and Pass by Reference in Javascript

Pass By Value

- In Pass by value, the function is called by directly passing the value of the variable as an argument. So any changes made inside the function do not affect the original value.
- In Pass by value, parameters passed as arguments create their **own copy.** So any changes made inside the function are made to the copied value not to the original value

```
function Passbyvalue(a, b) {
    let tmp;
    tmp = b;
    b = a;
    a = tmp;
    console.log(`Inside Pass by value
        function -> a = ${a} b = ${b}`);
}

let a = 1;
let b = 2;
console.log(`Before calling Pass by value
        Function -> a = ${a} b = ${b}`);

Passbyvalue(a, b);

console.log(`After calling Pass by value
       Function -> a =${a} b = ${b}`);

Before calling Pass by value
        Function -> a = 1 b = 2
Inside Pass by value
        function -> a = 2 b = 1
After calling Pass by value
       Function -> a =1 b = 2
```

Pass by Reference

- In Pass by Reference, Function is called by directly passing the reference/address of the variable as an argument. So changing the value inside the function also change the original value. In JavaScript **array and Object** follows pass by reference property.
- In Pass by reference, parameters passed as an arguments does not create its own copy, it refers to the original value so changes made inside function affect the original value.

```
function PassbyReference(obj) {
    let tmp = obj.a;
    obj.a = obj.b;
    obj.b = tmp;
    console.log(`Inside Pass By Reference
        Function -> a = ${obj.a} b = ${obj.b}`);
}

let obj = {
    a: 10,
    b: 20

}
console.log(`Before calling Pass By Reference
    Function -> a = ${obj.a} b = ${obj.b}`);

PassbyReference(obj)

console.log(`After calling Pass By Reference
    Function -> a = ${obj.a} b = ${obj.b}`);

Before calling Pass By Reference
    Function -> a = 10 b = 20
Inside Pass By Reference
        Function -> a = 20 b = 10
After calling Pass By Reference
    Function -> a = 20 b = 10
```

- ES7, ES8, and beyond: JS’s new features
- JS Design Patterns: Writing efficient code
- JS bundlers: Role in optimizing code for production
- Throttling, Debouncing: Controlling function execution rate
- Web APIs: Importance and working with them in JS
- Caching, Memoization: Performance Techniques
- Security best practices: XSS, CSRF protection
- ES modules: Role in modern JS development
- Prototypal vs. Classical Inheritance
- Error handling: try...catch blocks

1️⃣ JavaScript Engines: Learn about the engines that execute JavaScript, such as V8 (used in Chrome).

2️⃣ Value Types and Reference Types: Understand the distinction between primitive types and objects.

3️⃣ Primitive Types: Know the fundamental data types like string, number, boolean, null, undefined.

4️⃣ Expression vs Statement: Distinguish between expressions and statements in JavaScript code.

🔟 IIFE, Modules, and Namespaces: Organize and encapsulate code using Immediately Invoked Function Expressions and modularization.

1️⃣3️⃣ Factory Functions and Classes: Explore different approaches for creating objects.

1️⃣5️⃣ new, Constructor, instanceof, and Instances: Grasp object instantiation and constructor functions.

1️⃣6️⃣ Prototypal Nature : Utilize prototype-based inheritance for code efficiency.

1️⃣9️⃣ Pure Functions, Side Effects, State Mutation, and Event Propagation: Write clean and maintainable code.

2️⃣2️⃣ Recursion: Solve problems efficiently with recursive functions.

2️⃣3️⃣ Collections and Generators: Understand data structures and generators for data manipulation.

2️⃣5️⃣Partial Applications, Currying, Compose, and Pipe: Explore functional programming concepts for code modularity.

🔍 Describe the Function.prototype.bind method.

🔍 Explain the differences between feature detection, feature inference, and utilizing the User Agent (UA) string.

1️⃣1️⃣ Bitwise Operators, Type Arrays, and Array Buffers: Handle binary data efficiently with low-level operations.

- Unit testing: Popular frameworks like Jest or Mocha
- Data structures: Linked lists, stacks, queues in JS

7️⃣ and requestAnimationFrame: Manage time-related operations effectively.

2️⃣6️⃣ Data Structures & Algorithms: Explore various data structures for problem-solving.

Enumerate the benefits and drawbacks of immutability and explain how you can achieve it in your code.

Differentiate between synchronous and asynchronous functions and elucidate the event loop.

Explain how code sharing between files can be accomplished.

Explore how Object-Oriented Programming (OOP) principles can be applied when coding in JavaScript.

2️⃣7️⃣ Inheritance, Polymorphism, and Code Reuse: Master object-oriented programming concepts for scalability.

2️⃣8️⃣ Design Patterns: Apply reusable solutions to common problems for improved code architecture.

2️⃣9️⃣ Clean Code: Prioritize writing clean and readable code following best practices for maintainability.