JavaScript Interview Questions Guide

This document is a **comprehensive JavaScript interview preparation guide** containing a large collection of **carefully curated questions** with detailed explanations, real-world examples, and best practices.

It is designed to help learners build strong **JavaScript fundamentals and advanced knowledge**, enabling them to confidently approach interviews at **FAANG companies, unicorn startups, and top product-based firms**.

Who This Guide Is For

- Frontend developers preparing for interviews
- Backend engineers working with Node.js
- Developers transitioning into JavaScript from other languages
- Engineers looking to **level up their JavaScript expertise** and write cleaner, more efficient code

No.	Questions	Level
1	What are the various data types in JavaScript?	Basic
2	How do you check the data type of a variable?	Basic

No.	Questions	Level
3	What's the difference between a JavaScript variable that is: `null`, `undefined` or undeclared?	Basic
4	What are the differences between JavaScript variables created using `let`, `var` or `const`?	Basic
5	Why is it, in general, a good idea to leave the global JavaScript scope of a website as-is and never touch it?	Intermediate
6	How do you convert a string to a number in JavaScript?	Basic
7	What are template literals and how are they used?	Basic
8	Explain the concept of tagged templates	Intermediate
9	What is the spread operator and how is it used?	Basic
10	What are `Symbol`s used for in JavaScript?	Intermediate
11	What are proxies in JavaScript used for?	Advanced
12	Explain the concept of "hoisting" in JavaScript	Basic
13	Explain the difference in hoisting between `var`, `let`, and `const`	Basic
14	How does hoisting affect function declarations and expressions?	Advanced
15	What are the potential issues caused by hoisting?	Intermediate
16	How can you avoid problems related to hoisting?	Basic
17	What is the difference between `==` and `===` in JavaScript?	Basic
18	What language constructs do you use for iterating over object properties and array items in JavaScript?	Basic
19	What is the purpose of the `break` and `continue` statements?	Basic
20	What is the ternary operator and how is it used?	Basic

No.	Questions	Level
21	How do you access the index of an element in an array during iteration?	Basic
22	What is the purpose of the `switch` statement?	Basic
23	What are rest parameters and how are they used?	Basic
24	Explain the concept of the spread operator and its uses	Basic
25	What are the benefits of using spread syntax in JavaScript and how is it different from rest syntax?	Basic
26	What are iterators and generators in JavaScript and what are they used for?	Advanced
27	Explain the differences on the usage of `foo` between `function foo() {}` and `var foo = function() {}` in JavaScript	Basic
28	What is the difference between a parameter and an argument?	Basic
29	Explain the concept of hoisting with regards to functions	Basic
30	What's the difference between `.call` and `.apply` in JavaScript?	Basic
31	Can you offer a use case for the new arrow => function syntax?	Basic
32	Difference between: `function Person(){}`, `const person = Person()`, and `const person = new Person()` in JavaScript?	Basic
33	What is the definition of a higher-order function in JavaScript?	Basic
34	What are callback functions and how are they used?	Basic
35	What's a typical use case for anonymous functions in JavaScript?	Intermediate

No.	Questions	Level
36	What is recursion and how is it used in JavaScript?	Basic
37	What are default parameters and how are they used?	Basic
38	Explain why the following doesn't work as an IIFE: `function foo(){}();`. What needs to be changed to properly make it an IIFE?	Advanced
39	What are the various ways to create objects in JavaScript?	Basic
40	Explain the difference between dot notation and bracket notation for accessing object properties	Basic
41	What are the different methods for iterating over an array?	Basic
42	How do you add, remove, and update elements in an array?	Basic
43	What are the different ways to copy an object or an array?	Basic
44	Explain the difference between shallow copy and deep copy	Basic
45	What are the advantages of using the spread operator with arrays and objects?	Basic
46	How do you check if an object has a specific property?	Basic
47	Explain the difference between mutable and immutable objects in JavaScript	Intermediate
48	Explain the concept of destructuring assignment for objects and arrays	Basic
49	What is `Object.freeze()` for?	Intermediate
50	What is `Object.seal()` for?	Intermediate
51	What is `Object.preventExtensions()` for?	Intermediate
52	What are JavaScript object getters and setters for?	Intermediate
53	What are JavaScript object property flags and descriptors?	Advanced
54	How do you reliably determine whether an object is empty?	Basic

No.	Questions	Level
55	What is the event loop in JavaScript runtimes?	Basic
56	Explain the difference between synchronous and asynchronous functions in JavaScript	Basic
57	Explain the concept of a callback function in asynchronous operations	Basic
58	What are Promises and how do they work?	Basic
59	Explain the different states of a Promise	Intermediate
60	What are the pros and cons of using Promises instead of callbacks in JavaScript?	Intermediate
61	What is the use of `Promise.all()`	Basic
62	How is `Promise.all()` different from `Promise.allSettled()` ?	Intermediate
63	What is async/await and how does it simplify asynchronous code?	Intermediate
64	How do you handle errors in asynchronous operations?	Basic
65	Explain the concept of a microtask queue	Intermediate
66	What is the difference between `setTimeout()` , `setImmediate()` , and `process.nextTick()` ?	Intermediate
67	Explain how prototypal inheritance works in JavaScript	Basic
68	What is the prototype chain and how does it work?	Intermediate
69	Explain the difference between classical inheritance and prototypal inheritance	Basic
70	Explain the concept of inheritance in ES2015 classes	Basic
71	What is the purpose of the `new` keyword?	Basic
72	How do you create a constructor function?	Basic

No.	Questions	Level
73	What are the differences between JavaScript ES2015 classes and ES5 function constructors?	Basic
74	What advantage is there for using the JavaScript arrow syntax for a method in a constructor?	Basic
75	Why might you want to create static class members in JavaScript?	Intermediate
76	What is a closure in JavaScript, and how/why would you use one?	Intermediate
77	Explain the concept of lexical scoping	Basic
78	Explain the concept of scope in JavaScript	Basic
79	How can closures be used to create private variables?	Basic
80	What are the potential pitfalls of using closures?	Intermediate
81	Explain the difference between global scope, function scope, and block scope	Basic
82	Explain how `this` works in JavaScript	Basic
83	Explain `Function.prototype.bind` in JavaScript	Basic
84	Explain the different ways the `this` keyword can be bound	Intermediate
85	What are the common pitfalls of using the `this` keyword?	Basic
86	Explain the concept of `this` binding in event handlers	Basic
87	What is the DOM and how is it structured?	Basic
88	What's the difference between an "attribute" and a "property" in the DOM?	Intermediate
89	Explain the difference between `document.querySelector()` and `document.getElementById()`	Basic
90	How do you add, remove, and modify HTML elements using JavaScript?	Basic

No.	Questions	Level
91	What are event listeners and how are they used?	Basic
92	Explain the event phases in a browser	Intermediate
93	Describe event bubbling in JavaScript and browsers	Basic
94	Describe event capturing in JavaScript and browsers	Basic
95	Explain event delegation in JavaScript	Basic
96	How do you prevent the default behavior of an event?	Basic
97	What is the difference between `event.preventDefault()` and `event.stopPropagation()` ?	Intermediate
98	What is the difference between `mouseenter` and `mouseover` event in JavaScript and browsers?	Basic
99	What is the difference between `innerHTML` and `textContent` ?	Intermediate
100	How do you manipulate CSS styles using JavaScript?	Basic
101	Describe the difference between `<script>` , `<script async>` and `<script defer>`	Basic
102	What is the difference between the Window object and the Document object?	Intermediate
103	Describe the difference between a cookie, `sessionStorage` and `localStorage` in browsers	Basic
104	How do you make an HTTP request using the Fetch API?	Basic
105	What are the different ways to make an API call in JavaScript?	Basic
106	Explain AJAX in as much detail as possible	Basic
107	What are the advantages and disadvantages of using AJAX?	Basic
108	What are the differences between `XMLHttpRequest` and `fetch()` in JavaScript and browsers?	Basic

No.	Questions	Level
109	How do you abort a web request using `AbortController` in JavaScript?	Intermediate
110	Explain how JSONP works (and how it's not really Ajax)	Intermediate
111	What are workers in JavaScript used for?	Advanced
112	Explain the concept of the Web Socket API	Intermediate
113	What are JavaScript polyfills for?	Advanced
114	How do you detect if JavaScript is disabled on a page?	Intermediate
115	What is the `Intl` namespace object for?	Intermediate
116	How do you validate form elements using the Constraint Validation API?	Advanced
117	How do you use `window.history` API?	Basic
118	How do `<iframe>` on a page communicate?	Intermediate
119	Difference between document `load` event and document `DOMContentLoaded` event?	Intermediate
120	How do you redirect to a new page in JavaScript?	Basic
121	How do you get the query string values of the current page in JavaScript?	Basic
122	What are server-sent events?	Advanced
123	What are Progressive Web Applications (PWAs)?	Intermediate
124	What are modules and why are they useful?	Basic
125	Explain the differences between CommonJS modules and ES modules in JavaScript	Intermediate
126	How do you import and export modules in JavaScript?	Basic
127	What are the benefits of using a module bundler?	Intermediate

No.	Questions	Level
128	Explain the concept of tree shaking in module bundling	Intermediate
129	What are the metadata fields of a module?	Intermediate
130	What do you think of CommonJS vs ESM?	Basic
131	What are the different types of errors in JavaScript?	Intermediate
132	How do you handle errors using `try...catch` blocks?	Basic
133	What is the purpose of the `finally` block?	Basic
134	How can you create custom error objects?	Intermediate
135	Explain the concept of error propagation in JavaScript	Intermediate
136	What is currying and how does it work?	Intermediate
137	Explain the concept of partial application	Intermediate
138	What are the benefits of using currying and partial application?	Intermediate
139	Provide some examples of how currying and partial application can be used	Basic
140	How do currying and partial application differ from each other?	Intermediate
141	What are `Set` s and `Map` s and how are they used?	Basic
142	What are the differences between `Map` / `Set` and `WeakMap` / `WeakSet` in JavaScript?	Basic
143	How do you convert a `Set` to an array in JavaScript?	Basic
144	What is the difference between a `Map` object and a plain object in JavaScript?	Basic
145	How do `Set` s and `Map` s handle equality checks for objects?	Basic
146	What are some common performance bottlenecks in JavaScript applications?	Intermediate

No.	Questions	Level
147	Explain the concept of debouncing and throttling	Basic
148	How can you optimize DOM manipulation for better performance?	Advanced
149	What are some techniques for reducing reflows and repaints?	Advanced
150	Explain the concept of lazy loading and how it can improve performance	Basic
151	What are Web Workers and how can they be used to improve performance?	Advanced
152	Explain the concept of caching and how it can be used to improve performance	Basic
153	What are some tools that can be used to measure and analyze JavaScript performance?	Advanced
154	How can you optimize network requests for better performance?	Advanced
155	What are the different types of testing in software development?	Intermediate
156	Explain the difference between unit testing, integration testing, and end-to-end testing	Intermediate
157	What are some popular JavaScript testing frameworks?	Basic
158	How do you write unit tests for JavaScript code?	Intermediate
159	Explain the concept of test-driven development (TDD)	Intermediate
160	What are mocks and stubs and how are they used in testing?	Advanced
161	How can you test asynchronous code in JavaScript?	Intermediate
162	What are some best practices for writing maintainable and effective tests in JavaScript?	Intermediate

No.	Questions	Level
163	Explain the concept of code coverage and how it can be used to assess test quality	Intermediate
164	What are some tools that can be used for JavaScript testing?	Basic
165	What are design patterns and why are they useful?	Basic
166	Explain the concept of the Singleton pattern	Basic
167	What is the Factory pattern and how is it used?	Intermediate
168	Explain the Observer pattern and its use cases	Intermediate
169	What is the Module pattern and how does it help with encapsulation?	Intermediate
170	Explain the concept of the Prototype pattern	Basic
171	What is the Decorator pattern and how is it used?	Intermediate
172	Explain the concept of the Strategy pattern	Intermediate
173	What is the Command pattern and how is it used?	Intermediate
174	Why is extending built-in JavaScript objects not a good idea?	Intermediate
175	What is Cross-Site Scripting (XSS) and how can you prevent it?	Intermediate
176	Explain the concept of Cross-Site Request Forgery (CSRF) and its mitigation techniques	Intermediate
177	How can you prevent SQL injection vulnerabilities in JavaScript applications?	Intermediate
178	What are some best practices for handling sensitive data in JavaScript?	Intermediate
179	Explain the concept of Content Security Policy (CSP) and how it enhances security	Intermediate
180	What are some common security headers and their purpose?	Intermediate

No.	Questions	Level
181	How can you prevent clickjacking attacks?	Advanced
182	Explain the concept of input validation and its importance in security	Intermediate
183	What are some tools and techniques for identifying security vulnerabilities in JavaScript code?	Intermediate
184	How can you implement secure authentication and authorization in JavaScript applications?	Advanced
185	Explain the same-origin policy with regards to JavaScript	Intermediate
186	What is `'use strict';` in JavaScript for?	Advanced
187	What tools and techniques do you use for debugging JavaScript code?	Intermediate
188	How does JavaScript garbage collection work?	Advanced
189	Explain what a single page app is and how to make one SEO-friendly	Intermediate
190	How can you share code between JavaScript files?	Basic
191	How do you organize your code?	Intermediate
192	What are some of the advantages/disadvantages of writing JavaScript code in a language that compiles to JavaScript?	Advanced
193	When would you use `document.write()`?	Advanced

What are the various data types in JavaScript?

In JavaScript, data types can be categorized into `primitive` and `non-primitive` types:

Primitive data types

- **Number**: Represents both integers and floating-point numbers.
- **String**: Represents sequences of characters.

- **Boolean**: Represents `true` or `false` values.
- **Undefined**: A variable that has been declared but not assigned a value.
- **Null**: Represents the intentional absence of any object value.
- **Symbol**: A unique and immutable value used as object property keys.
- **BigInt**: Represents integers with arbitrary precision.

Non-primitive (Reference) data types

- **Object**: Used to store collections of data.
- **Array**: An ordered collection of data.
- **Function**: A callable object.
- **Date**: Represents dates and times.
- **RegExp**: Represents regular expressions.
- **Map**: A collection of keyed data items.
- **Set**: A collection of unique values.

The primitive types store a single value, while non-primitive types can store collections of data or complex entities.

How do you check the data type of a variable?

To check the data type of a variable in JavaScript, you can use the `typeof` operator. For example, `typeof variableName` will return a string indicating the type of the variable, such as `"string"`, `"number"`, `"boolean"`, `"object"`, `"function"`, `"undefined"`, or `"symbol"`. For arrays and `null`, you can use `Array.isArray(variableName)` and `variableName === null`, respectively.

What's the difference between a JavaScript variable that is: `null`, `undefined` or undeclared?

Trait	`null`	`undefined`	Undeclared
Meaning	Explicitly set by the developer to indicate that a variable has no value	Variable has been declared but not assigned a value	Variable has not been declared at all

Trait	`null`	`undefined`	Undeclared
Type (via `typeof` operator)	`'object'`	`'undefined'`	`'undefined'`
Equality Comparison	`null == undefined` is `true`	`undefined == null` is `true`	Throws a `ReferenceError`

What are the differences between JavaScript variables created using `let`, `var` or `const`?

In JavaScript, `let`, `var`, and `const` are all keywords used to declare variables, but they differ significantly in terms of scope, initialization rules, whether they can be redeclared or reassigned and the behavior when they are accessed before declaration:

Behavior	`var`	`let`	`const`
Scope	Function or Global	Block	Block
Initialization	Optional	Optional	Required
Redeclaration	Yes	No	No
Reassignment	Yes	Yes	No
Accessing before declaration	`undefined`	`ReferenceError`	`ReferenceError`

Why is it, in general, a good idea to leave the global JavaScript scope of a website as-is and never touch it?

JavaScript that is executed in the browser has access to the global scope (the `window` object). In general it's a good software engineering practice to not pollute the global namespace unless you are working on a feature that truly needs to be global – it is needed by the entire page. Several reasons to avoid touching the global scope:

- **Naming conflicts**: Sharing the global scope across scripts can cause conflicts and bugs when new global variables or changes are introduced.

- **Cluttered global namespace**: Keeping the global namespace minimal avoids making the codebase hard to manage and maintain.
- **Scope leaks**: Unintentional references to global variables in closures or event handlers can cause memory leaks and performance issues.
- **Modularity and encapsulation**: Good design promotes keeping variables and functions within their specific scopes, enhancing organization, reusability, and maintainability.
- **Security concerns**: Global variables are accessible by all scripts, including potentially malicious ones, posing security risks, especially if sensitive data is stored there.
- **Compatibility and portability**: Heavy reliance on global variables reduces code portability and integration ease with other libraries or frameworks.

Follow these best practices to avoid global scope pollution:

- **Use local variables**: Declare variables within functions or blocks using `var`, `let`, or `const` to limit their scope.
- **Pass variables as function parameters**: Maintain encapsulation by passing variables as parameters instead of accessing them globally.
- **Use immediately invoked function expressions (IIFE)**: Create new scopes with IIFEs to prevent adding variables to the global scope.
- **Use modules**: Encapsulate code with module systems to maintain separate scopes and manageability.

How do you convert a string to a number in JavaScript?

In JavaScript, you can convert a string to a number using several methods. The most common ones are `Number()`, `parseInt()`, `parseFloat()`, and the unary plus operator (`+`). For example, `Number("123")` converts the string `"123"` to the number `123`, and `parseInt("123.45")` converts the string `"123.45"` to the integer `123`.

What are template literals and how are they used?

Template literals are a feature in JavaScript that allow for easier string interpolation and multi-line strings. They are enclosed by backticks (`` ` ``) instead of single or double quotes. You can embed expressions within template literals using `${expression}` syntax.

Example:

```
const myName = 'John';
const greeting = `Hello, ${myName}!`;
console.log(greeting); // Output: Hello, John!
```

Explain the concept of tagged templates

Tagged templates in JavaScript allow you to parse template literals with a function. The function receives the literal strings and the values as arguments, enabling custom processing of the template. For example:

```
function tag(strings, ...values) {
  return strings[0] + values[0] + strings[1] + values[1] + strings[2];
}

const result = tag`Hello ${'world'}! How are ${'you'}?`;
console.log(result); // "Hello world! How are you?"
```

What is the spread operator and how is it used?

The spread operator, represented by three dots (`...`), is used in JavaScript to expand iterable objects like arrays or strings into individual elements. It can also be used to spread object properties. For example, you can use it to combine arrays, copy arrays, or pass array elements as arguments to a function.

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

const obj1 = { a: 1, b: 2 };
const obj2 = { c: 3, d: 4 };
const combinedObj = { ...obj1, ...obj2 };
console.log(combinedObj); // { a: 1, b: 2, c: 3, d: 4 }
```

What are `Symbol`s used for in JavaScript?

`Symbol`s in JavaScript are a new primitive data type introduced in ES6 (ECMAScript 2015). They are unique and immutable identifiers that is primarily for object property keys to avoid name collisions. These values can be created using `Symbol(...)` function, and each `Symbol` value is guaranteed to be unique, even if they have the same key/description. `Symbol` properties are not enumerable in `for...in` loops or `Object.keys()`, making them suitable for creating private/internal object state.

```
let sym1 = Symbol();
let sym2 = Symbol('myKey');

console.log(typeof sym1); // "symbol"
console.log(sym1 === sym2); // false, because each symbol is unique

let obj = {};
let sym = Symbol('uniqueKey');

obj[sym] = 'value';
console.log(obj[sym]); // "value"
```

Note: The `Symbol()` function must be called without the `new` keyword. It is not exactly a constructor because it can only be called as a function instead of with `new Symbol()`.

What are proxies in JavaScript used for?

In JavaScript, a proxy is an object that acts as an intermediary between an object and the code. Proxies are used to intercept and customize the fundamental operations of JavaScript objects, such as property access, assignment, function invocation, and more.

Here's a basic example of using a `Proxy` to log every property access:

```
const myObject = {
  name: 'John',
  age: 42,
};

const handler = {
  get: function (target, prop, receiver) {
    console.log(`Someone accessed property "${prop}"`);
    return target[prop];
  },
};

const proxiedObject = new Proxy(myObject, handler);

console.log(proxiedObject.name);
// Someone accessed property "name"
// 'John'

console.log(proxiedObject.age);
// Someone accessed property "age"
// 42
```

Use cases include:

- **Property access interception**: Intercept and customize property access on an object.
  - **Property assignment validation**: Validate property values before they are set on the target object.
  - **Logging and debugging**: Create wrappers for logging and debugging interactions with an object
  - **Creating reactive systems**: Trigger updates in other parts of your application when object properties change (data binding).
  - **Data transformation**: Transforming data being set or retrieved from an object.
  - **Mocking and stubbing in tests**: Create mock or stub objects for testing purposes, allowing you to isolate dependencies and focus on the unit under test
- **Function invocation interception**: Used to cache and return the result of frequently accessed methods if they involve network calls or computationally intensive logic, improving performance
- **Dynamic property creation**: Useful for defining properties on-the-fly with default values and avoid storing redundant data in objects.

Explain the concept of “hoisting” in JavaScript

Hoisting is a JavaScript mechanism where variable and function declarations are moved (“hoisted”) to the top of their containing scope during the compile phase.

- **Variable declarations (`var`)**: Declarations are hoisted, but not initializations. The value of the variable is `undefined` if accessed before initialization.
- **Variable declarations (`let` and `const`)**: Declarations are hoisted, but not initialized. Accessing them results in `ReferenceError` until the actual declaration is encountered.
- **Function expressions (`var`)**: Declarations are hoisted, but not initializations. The value of the variable is `undefined` if accessed before initialization.

- **Function declarations (`function`)**: Both declaration and definition are fully hoisted.
- **Class declarations (`class`)**: Declarations are hoisted, but not initialized. Accessing them results in `ReferenceError` until the actual declaration is encountered.
- **Import declarations (`import`)**: Declarations are hoisted, and side effects of importing the module are executed before the rest of the code.

The following behavior summarizes the result of accessing the variables before they are declared.

Declaration	Accessing before declaration
`var foo`	`undefined`
`let foo`	`ReferenceError`
`const foo`	`ReferenceError`
`class Foo`	`ReferenceError`
`var foo = function() { ... }`	`undefined`
`function foo() { ... }`	Normal
`import`	Normal

Explain the difference in hoisting between `var` , `let` , and `const`

`var` declarations are hoisted to the top of their scope and initialized with `undefined`, allowing them to be used before their declaration. `let` and `const` declarations are also hoisted but are not initialized, resulting in a `ReferenceError` if accessed before their declaration. `const` additionally requires an initial value at the time of declaration.

How does hoisting affect function declarations and expressions?

Hoisting in JavaScript means that function declarations are moved to the top of their containing scope during the compile phase, making them available throughout the entire scope. This allows you to call a function before it is defined in the code. However, function expressions are not hoisted in the same way. If you try to call a function expression before it is defined, you will get an error because the variable holding the function is hoisted but not its assignment.

```
// Function declaration
console.log(foo()); // Works fine
function foo() {
  return 'Hello';
}

// Function expression
console.log(bar()); // Throws TypeError: bar is not a function
var bar = function () {
  return 'Hello';
};
```

What are the potential issues caused by hoisting?

Hoisting can lead to unexpected behavior in JavaScript because variable and function declarations are moved to the top of their containing scope during the compilation phase. This can result in `undefined` values for variables if they are used before their declaration and can cause confusion with function declarations and expressions. For example:

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

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

How can you avoid problems related to hoisting?

To avoid problems related to hoisting, always declare variables at the top of their scope using `let` or `const` instead of `var`. This ensures that variables are block-scoped and not hoisted to the top of their containing function or global scope. Additionally, declare functions before they are called to avoid issues with function hoisting.

```
// Use let or const
let x = 10;
const y = 20;
console.log(x, y); // Output: 10 20
```

```
// Declare functions before calling them
function myFunction() {
  console.log('Hello, world!');
}
myFunction(); // Output: 'Hello, world!'
```

What is the difference between `==` and `===` in JavaScript?

`==` is the abstract equality operator while `===` is the strict equality operator. The `==` operator will compare for equality after doing any necessary type conversions. The `===` operator will not do type conversion, so if two values are not the same type `===` will simply return `false`.

Operator	`==`	`===`
Name	(Loose) Equality operator	Strict equality operator
Type coercion	Yes	No
Compares value and type	No	Yes

What language constructs do you use for iterating over object properties and array items in JavaScript?

There are multiple ways to iterate over object properties as well as arrays in JavaScript:

`for...in` loop

The `for...in` loop iterates over all enumerable properties of an object, including inherited enumerable properties. So it is important to have a check if you only want to iterate over object's own properties

```
const obj = {
  a: 1,
  b: 2,
  c: 3,
};

for (const key in obj) {
  // To avoid iterating over inherited properties
  if (Object.hasOwn(obj, key)) {
    console.log(`${key}: ${obj[key]}`);
  }
}
```

`Object.keys()`

`Object.keys()` returns an array of the object's own enumerable property names. You can then use a for...of loop or forEach to iterate over this array.

```
const obj = {
  a: 1,
  b: 2,
  c: 3,
};

Object.keys(obj).forEach((key) => {
  console.log(`${key}: ${obj[key]}`);
});
```

Most common ways to iterate over array are using `for` loop and `Array.prototype.forEach` method.

Using `for` loop

```
let array = [1, 2, 3, 4, 5, 6];
for (let index = 0; index < array.length; index++) {
  console.log(array[index]);
}
```

Using `Array.prototype.forEach` method

```
let array = [1, 2, 3, 4, 5, 6];
array.forEach((number, index) => {
  console.log(`${number} at index ${index}`);
});
```

Using `for...of`

This method is the newest and most convenient way to iterate over arrays. It automatically iterates over each element without requiring you to manage the index.

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

for (const number of numbers) {
  console.log(number);
}
```

There are also other inbuilt methods available which are suitable for specific scenarios for example:

- `Array.prototype.filter` : You can use the `filter` method to create a new array containing only the elements that satisfy a certain condition.
- `Array.prototype.map` : You can use the `map` method to create a new array based on the existing one, transforming each element with a provided function.
- `Array.prototype.reduce` : You can use the `reduce` method to combine all elements into a single value by repeatedly calling a function that takes two arguments: the accumulated value and the current element.

What is the purpose of the `break` and `continue` statements?

The `break` statement is used to exit a loop or switch statement prematurely, while the `continue` statement skips the current iteration of a loop and proceeds to the next iteration. For example, in a `for` loop, `break` will stop the loop entirely, and `continue` will skip to the next iteration.

```
for (let i = 0; i < 10; i++) {
  if (i === 5) break; // exits the loop when i is 5
  console.log(i);
}

for (let i = 0; i < 10; i++) {
  if (i === 5) continue; // skips the iteration when i is 5
  console.log(i);
}
```

What is the ternary operator and how is it used?

The ternary operator is a shorthand for an `if-else` statement in JavaScript. It takes three operands: a condition, a result for true, and a result for false. The syntax is `condition ? expr1 : expr2`. For example, `let result = (a > b) ? 'a is` `greater' : 'b is greater';` assigns `'a is greater'` to `result` if `a` is greater than `b`, otherwise it assigns `'b is greater'`.

How do you access the index of an element in an array during iteration?

To access the index of an element in an array during iteration, you can use methods like `forEach`, `map`, `for...of` with `entries`, or a traditional `for` loop. For example, using `forEach`:

```
const array = ['a', 'b', 'c'];
array.forEach((element, index) => {
  console.log(index, element);
});
```

What is the purpose of the `switch` statement?

The `switch` statement is used to execute one block of code among many based on the value of an expression. It is an alternative to using multiple `if...else if` statements. The `switch` statement evaluates an expression, matches the expression's value to a `case` label, and executes the associated block of code. If no `case` matches, the `default` block is executed.

```
switch (expression) {
  case value1:
    // code to be executed if expression === value1
    break;
  case value2:
    // code to be executed if expression === value2
    break;
  default:
  // code to be executed if no case matches
}
```

What are rest parameters and how are they used?

Rest parameters in JavaScript allow a function to accept an indefinite number of arguments as an array. They are denoted by three dots (`...`) followed by the name of the array. This feature is useful for functions that need to handle multiple arguments without knowing the exact number in advance.

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

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

Explain the concept of the spread operator and its uses

The spread operator (`...`) in JavaScript allows you to expand elements of an iterable (like an array or object) into individual elements. It is commonly used for copying arrays or objects, merging arrays or objects, and passing elements of an array as arguments to a function.

```
// Copying an array
const arr1 = [1, 2, 3];
const arr2 = [...arr1];
console.log(arr2); // Output: [1, 2, 3]

// Merging arrays
const arr3 = [4, 5, 6];
const mergedArray = [...arr1, ...arr3];
console.log(mergedArray); // Output: [1, 2, 3, 4, 5, 6]

// Copying an object
const obj1 = { a: 1, b: 2 };
const obj2 = { ...obj1 };
console.log(obj2); // Output: { a: 1, b: 2 }

// Merging objects
const obj3 = { c: 3, d: 4 };
const mergedObject = { ...obj1, ...obj3 };
console.log(mergedObject); // Output: { a: 1, b: 2, c: 3, d: 4 }

// Passing array elements as function arguments
const sum = (x, y, z) => x + y + z;
const numbers = [1, 2, 3];
console.log(sum(...numbers)); // Output: 6
```

What are the benefits of using spread syntax in JavaScript and how is it different from rest syntax?

Spread syntax (`...`) allows an iterable (like an array or string) to be expanded into individual elements. This is often used as a convenient and modern way to create new arrays or objects by combining existing ones.

Operation	Traditional	Spread
Array cloning	`arr.slice()`	`[...arr]`
Array merging	`arr1.concat(arr2)`	`[...arr1, ...arr2]`
Object cloning	`Object.assign({}, obj)`	`{ ...obj }`
Object merging	`Object.assign({}, obj1, obj2)`	`{ ...obj1, ...obj2 }`

Rest syntax is the opposite of what spread syntax does. It collects a variable number of arguments into an array. This is often used in function parameters to handle a dynamic number of arguments.

```
// Using rest syntax in a function
function sum(...numbers) {
  return numbers.reduce((total, num) => total + num, 0);
}

console.log(sum(1, 2, 3)); // Output: 6
```

What are iterators and generators in JavaScript and what are they used for?

In JavaScript, iterators and generators are powerful tools for managing sequences of data and controlling the flow of execution in a more flexible way.

Iterators are objects that define a sequence and potentially a return value upon its termination. It adheres to a specific interface:

- An iterator object must implement a `next()` method.
- The `next()` method returns an object with two properties:
  - `value` : The next value in the sequence.
  - `done` : A boolean that is `true` if the iterator has finished its sequence, otherwise `false` .

Here's an example of an object implementing the iterator interface.

```
const iterator = {
  current: 0,
  last: 5,
  next() {
    if (this.current <= this.last) {
      return { value: this.current++, done: false };
    } else {
      return { value: undefined, done: true };
    }
  },
};

let result = iterator.next();
while (!result.done) {
  console.log(result.value); // Logs 0, 1, 2, 3, 4, 5
  result = iterator.next();
}
```

Generators are a special functions that **can pause execution and resume at a later point**. It uses the `function*` syntax and the `yield` keyword to control the flow of execution. When you call a generator function, it doesn't execute completely like a regular function. Instead, it returns an iterator object. Calling the `next()` method on the returned iterator advances the generator to the next `yield` statement, and the value after `yield` becomes the return value of `next()`.

```
function* numberGenerator() {
  let num = 0;
  while (num <= 5) {
    yield num++;
  }
}

const gen = numberGenerator();
console.log(gen.next()); // { value: 0, done: false }
console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
console.log(gen.next()); // { value: 3, done: false }
console.log(gen.next()); // { value: 4, done: false }
console.log(gen.next()); // { value: 5, done: false }
console.log(gen.next()); // { value: undefined, done: true }
```

Generators are powerful for creating iterators on-demand, especially for infinite sequences or complex iteration logic. They can be used for:

- Lazy evaluation – processing elements only when needed, improving memory efficiency for large datasets.
- Implementing iterators for custom data structures.
- Creating asynchronous iterators for handling data streams.

Explain the differences on the usage of `foo` between `function foo() {}` and `var foo = function() {}` in JavaScript

`function foo() {}` a function declaration while the `var foo = function() {}` is a function expression. The key difference is that function declarations have its body hoisted but the bodies of function expressions are not (they have the same hoisting behavior as `var`-declared variables).

If you try to invoke a function expression before it is declared, you will get an `Uncaught TypeError: XXX is not a function` error.

Function declarations can be called in the enclosing scope even before they are declared.

```
foo(); // 'FOOOOO'
function foo() {
  console.log('FOOOOO');
}
```

Function expressions if called before they are declared will result in an error.

```
foo(); // Uncaught TypeError: foo is not a function
var foo = function () {
  console.log('FOOOOO');
};
```

Another key difference is in the scope of the function name. Function expressions can be named by defining it after the `function` and before the parenthesis. However when using named function expressions, the function name is only accessible within the function itself. Trying to access it outside will result in an error or `undefined`.

```
const myFunc = function namedFunc() {
  console.log(namedFunc); // Works
};
```

```
myFunc(); // Runs the function and logs the function reference
console.log(namedFunc); // ReferenceError: namedFunc is not defined
```

Note: The examples uses `var` due to legacy reasons. Function expressions can be defined using `let` and `const` and the key difference is in the hoisting behavior of those keywords.

What is the difference between a parameter and an argument?

A parameter is a variable in the declaration of a function, while an argument is the actual value passed to the function when it is called. For example, in the function `function add(a, b) { return a + b; }`, `a` and `b` are parameters. When you call `add(2, 3)`, `2` and `3` are arguments.

Explain the concept of hoisting with regards to functions

Hoisting in JavaScript is a behavior where function declarations are moved to the top of their containing scope during the compile phase. This means you can call a function before it is defined in the code. However, this does not apply to function expressions or arrow functions, which are not hoisted in the same way.

```
// Function declaration
hoistedFunction(); // Works fine
function hoistedFunction() {
  console.log('This function is hoisted');
}

// Function expression
nonHoistedFunction(); // Throws an error
var nonHoistedFunction = function () {
  console.log('This function is not hoisted');
};
```

What's the difference between `.call` and `.apply` in JavaScript?

`.call` and `.apply` are both used to invoke functions with a specific `this` context and arguments. The primary difference lies in how they accept arguments:

- `.call(thisArg, arg1, arg2, ...)`: Takes arguments individually.

- `.apply(thisArg, [argsArray])`: Takes arguments as an array.

Assuming we have a function `add`, the function can be invoked using `.call` and `.apply` in the following manner:

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

console.log(add.call(null, 1, 2)); // 3
console.log(add.apply(null, [1, 2])); // 3
```

Can you offer a use case for the new arrow => function syntax?

Arrow functions provide a concise syntax for writing functions in JavaScript. They are particularly useful for maintaining the `this` context within methods and callbacks. For example, in an event handler or array method like `map`, arrow functions can simplify the code and avoid issues with `this` binding.

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

Difference between: `function Person(){}`, `const person = Person()`, and `const person = new Person()` in JavaScript?

- `function Person(){}`: A function declaration in JavaScript. It can be used as a regular function or as a constructor.
- `const person = Person()`: Calls `Person` as a regular function, not a constructor. If `Person` is intended to be a constructor, this will lead to unexpected behavior.
- `const person = new Person()`: Creates a new instance of `Person`, correctly utilizing the constructor function to initialize the new object.

Aspect	`function Person(){}`	`const person = Person()`	`const person = new Person()`
Type	Function declaration	Function call	Constructor call

Aspect	`function Person(){}`	`const person = Person()`	`const person = new Person()`
Usage	Defines a function	Invokes `Person` as a regular function	Creates a new instance of `Person`
Instance Creation	No instance created	No instance created	New instance created
Common Mistake	N/A	Misusing as constructor leading to `undefined`	None (when used correctly)

What is the definition of a higher-order function in JavaScript?

A higher-order function is any function that takes one or more functions as arguments, which it uses to operate on some data, and/or returns a function as a result.

Higher-order functions are meant to abstract some operation that is performed repeatedly. The classic example of this is `Array.prototype.map()`, which takes an array and a function as arguments. `Array.prototype.map()` then uses this function to transform each item in the array, returning a new array with the transformed data. Other popular examples in JavaScript are `Array.prototype.forEach()`, `Array.prototype.filter()`, and `Array.prototype.reduce()`. A higher-order function doesn't just need to be manipulating arrays as there are many use cases for returning a function from another function. `Function.prototype.bind()` is an example that returns another function.

Imagine a scenario where we have an array of names that we need to transform to uppercase. The imperative way will be as such:

```
const names = ['irish', 'daisy', 'anna'];

function transformNamesToUppercase(names) {
  const results = [];
  for (let i = 0; i < names.length; i++) {
    results.push(names[i].toUpperCase());
  }
  return results;
}
```

```
console.log(transformNamesToUppercase(names));
// ['IRISH', 'DAISY', 'ANNA']
```

Using `Array.prototype.map(transformerFn)` makes the code shorter and more declarative.

```
const names = ['irish', 'daisy', 'anna'];

function transformNamesToUppercase(names) {
  return names.map((name) => name.toUpperCase());
}

console.log(transformNamesToUppercase(names));
// ['IRISH', 'DAISY', 'ANNA']
```

What are callback functions and how are they used?

A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action. They are commonly used for asynchronous operations like handling events, making API calls, or reading files. For example:

```
function fetchData(callback) {
  // assume an asynchronous operation to fetch data
  const data = { name: 'John Doe' };
  callback(data);
}

function handleData(data) {
  console.log(data);
}

fetchData(handleData);
```

What’s a typical use case for anonymous functions in JavaScript?

Anonymous function in Javascript is a function that does not have any name associated with it. They are typically used as arguments to other functions or assigned to variables.

```
const arr = [-1, 0, 5, 6];
```

```
// The filter method is passed an anonymous function.
arr.filter((x) => x > 1); // [5, 6]
```

They are often used as arguments to other functions, known as higher-order functions, which can take functions as input and return a function as output. Anonymous functions can access variables from the outer scope, a concept known as closures, allowing them to "close over" and remember the environment in which they were created.

```
// Encapsulating Code
(function () {
  // Some code here.
})();

// Callbacks
setTimeout(function () {
  console.log('Hello world!');
}, 1000);

// Functional programming constructs
const arr = [1, 2, 3];
const double = arr.map(function (el) {
  return el * 2;
});
console.log(double); // [2, 4, 6]
```

What is recursion and how is it used in JavaScript?

Recursion is a programming technique where a function calls itself to solve a problem. In JavaScript, recursion is used to solve problems that can be broken down into smaller, similar sub-problems. A base case is essential to stop the recursive calls and prevent infinite loops. For example, calculating the factorial of a number can be done using recursion:

```
function factorial(n) {
  if (n === 0) {
    return 1;
  }
  return n * factorial(n - 1);
}

console.log(factorial(4)); // Output: 24
```

What are default parameters and how are they used?

Default parameters in JavaScript allow you to set default values for function parameters if no value or `undefined` is passed. This helps avoid `undefined` values and makes your code more robust. You can define default parameters by assigning a value to the parameter in the function definition.

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

greet(); // Output: Hello, Guest!
greet('Alice'); // Output: Hello, Alice!
```

Explain why the following doesn't work as an IIFE: `function foo(){}();`. What needs to be changed to properly make it an IIFE?

The code `function foo(){}();` doesn't work as an Immediately Invoked Function Expression (IIFE) because the JavaScript parser treats `function foo(){}` as a function declaration, not an expression. To make it an IIFE, you need to wrap the function in parentheses to turn it into a function expression: `(function foo(){})();`.

What are the various ways to create objects in JavaScript?

Creating objects in JavaScript offers several methods:

- **Object literals (`{}`)**: Simplest and most popular approach. Define key-value pairs within curly braces.
- **`Object()` constructor**: Use `new Object()` with dot notation to add properties.
- **`Object.create()`**: Create new objects using existing objects as prototypes, inheriting properties and methods.
- **Constructor functions**: Define blueprints for objects using functions, creating instances with `new`.
- **ES2015 classes**: Structured syntax similar to other languages, using `class` and `constructor` keywords.

Explain the difference between dot notation and bracket notation for accessing object properties

Dot notation and bracket notation are two ways to access properties of an object in JavaScript. Dot notation is more concise and readable but can only be used with valid JavaScript identifiers. Bracket notation is more flexible and can be used with property names that are not valid identifiers, such as those containing spaces or special characters.

```
const obj = { name: 'Alice', 'favorite color': 'blue' };

// Dot notation
console.log(obj.name); // Alice

// Bracket notation
console.log(obj['favorite color']); // blue
```

What are the different methods for iterating over an array?

There are several methods to iterate over an array in JavaScript. The most common ones include `for` loops, `forEach`, `map`, `filter`, `reduce`, and `for...of`. Each method has its own use case. For example, `for` loops are versatile and can be used for any kind of iteration, while `forEach` is specifically for executing a function on each array element. `map` is used for transforming arrays, `filter` for filtering elements, `reduce` for accumulating values, and `for...of` for iterating over iterable objects.

How do you add, remove, and update elements in an array?

To add elements to an array, you can use methods like `push`, `unshift`, or `splice`. To remove elements, you can use `pop`, `shift`, or `splice`. To update elements, you can directly access the array index and assign a new value.

```
let arr = [1, 2, 3];

// Add elements
arr.push(4); // [1, 2, 3, 4]
arr.unshift(0); // [0, 1, 2, 3, 4]
arr.splice(2, 0, 1.5); // [0, 1, 1.5, 2, 3, 4]

// Remove elements
arr.pop(); // [0, 1, 1.5, 2, 3]
arr.shift(); // [1, 1.5, 2, 3]
arr.splice(1, 1); // [1, 2, 3]
```

```
// Update elements
arr[1] = 5; // [1, 5, 3]
console.log(arr); // Final state: [1, 5, 3]
```

Note: If you try to `console.log(arr)` after each operation in some environments (like Chrome DevTools), you may only see the final state of `arr`. This happens because the console sometimes keeps a live reference to the array instead of logging its state at the exact moment. To see intermediate states properly, store snapshots using `console.log([...arr])` or print values immediately after each operation.

What are the different ways to copy an object or an array?

To copy an object or an array in JavaScript, you can use several methods. For shallow copies, you can use the spread operator (`...`) or `Object.assign()`. For deep copies, you can use `JSON.parse(JSON.stringify())` or libraries like Lodash's `_.cloneDeep()`.

```
// Shallow copy of an array
const originalArray = [1, 2, 3];
const shallowCopyArray = [...originalArray];
console.log(shallowCopyArray); // [1, 2, 3]

// Shallow copy of an object
const originalObject = { a: 1, b: 2 };
const shallowCopyObject = { ...originalObject };
console.log(shallowCopyObject); // { a: 1, b: 2 };

// Deep copy using JSON methods
const deepCopyObject = JSON.parse(JSON.stringify(originalObject));
console.log(deepCopyObject); // { a: 1, b: 2 };
```

Explain the difference between shallow copy and deep copy

A shallow copy duplicates the top-level properties of an object, but nested objects are still referenced. A deep copy duplicates all levels of an object, creating entirely new instances of nested objects. For example, using `Object.assign()` creates a shallow copy, while using libraries like `Lodash` or `structuredClone()` in modern JavaScript can create deep copies.

```
// Shallow copy example
let obj1 = { a: 1, b: { c: 2 } };
let shallowCopy = Object.assign({}, obj1);
shallowCopy.b.c = 3;
console.log(shallowCopy.b.c); // Output: 3
console.log(obj1.b.c); // Output: 3 (original nested object changed too!)

// Deep copy example
let obj2 = { a: 1, b: { c: 2 } };
let deepCopy = JSON.parse(JSON.stringify(obj2));
deepCopy.b.c = 4;
console.log(deepCopy.b.c); // Output: 4
console.log(obj2.b.c); // Output: 2
//(original nested object remains unchanged)
```

What are the advantages of using the spread operator with arrays and objects?

The spread operator (`...`) in JavaScript allows you to easily copy arrays and objects, merge them, and add new elements or properties. It simplifies syntax and improves readability. For arrays, it can be used to concatenate or clone arrays. For objects, it can be used to merge objects or add new properties.

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

// Objects
const obj1 = { a: 1, b: 2 };
const obj2 = { ...obj1, c: 3 };
console.log(obj2); // { a: 1, b: 2, c: 3 }
```

How do you check if an object has a specific property?

To check if an object has a specific property, you can use the `in` operator or the `hasOwnProperty` method. The `in` operator checks for both own and inherited properties, while `hasOwnProperty` checks only for own properties.

```
const obj = { key: 'value' };

// Using the `in` operator
if ('key' in obj) {
  console.log('Property exists');
}

// Using `hasOwnProperty`
if (obj.hasOwnProperty('key')) {
  console.log('Property exists');
}
```

Explain the difference between mutable and immutable objects in JavaScript

Mutable objects allow for modification of properties and values after creation, which is the default behavior for most objects.

```
const mutableObject = {
  name: 'John',
  age: 30,
};

// Modify the object
mutableObject.name = 'Jane';

// The object has been modified
console.log(mutableObject); // Output: { name: 'Jane', age: 30 }
```

Immutable objects cannot be directly modified after creation. Its content cannot be changed without creating an entirely new value.

```
const immutableObject = Object.freeze({
  name: 'John',
  age: 30,
});

// Attempt to modify the object
immutableObject.name = 'Jane';

// The object remains unchanged
console.log(immutableObject); // Output: { name: 'John', age: 30 }
```

The key difference between mutable and immutable objects is modifiability. Immutable objects cannot be modified after they are created, while mutable objects can be.

Explain the concept of destructuring assignment for objects and arrays

Destructuring assignment is a syntax in JavaScript that allows you to unpack values from arrays or properties from objects into distinct variables. For arrays, you use square brackets, and for objects, you use curly braces. For example:

```
// Array destructuring
const [a, b] = [1, 2];

// Object destructuring
const { name, age } = { name: 'John', age: 30 };
```

What is `Object.freeze()` for?

`Object.freeze()` is used to make an object immutable. Once an object is frozen, you cannot add, remove, or modify its properties. This is useful for creating constants or ensuring that an object remains unchanged throughout the program.

```
const obj = { name: 'John' };
Object.freeze(obj);
obj.name = 'Doe'; // This will not change the name property

console.log(obj); // { name: 'John' }
```

What is `Object.seal()` for?

`Object.seal()` is used to prevent new properties from being added to an object and to mark all existing properties as non-configurable. This means you can still modify the values of existing properties, but you cannot delete them or add new ones. Doing so will throw errors in strict mode but fail silently in non-strict mode. In the following examples, you can uncomment the ‘use strict’ comment to see this.

```
// 'use strict'

const obj = { name: 'John' };
Object.seal(obj);

obj.name = 'Jane'; // Allowed
obj.age = 30; // Not allowed, throws an error in strict mode
delete obj.name; // Not allowed, throws an error in strict mode
```

```
console.log(obj); // { name: 'Jane } (unchanged)
```

What is `Object.preventExtensions()` for?

`Object.preventExtensions()` is a method in JavaScript that prevents new properties from being added to an object. However, it does not affect the deletion or modification of existing properties. This method is useful when you want to ensure that an object remains in a certain shape and no additional properties can be added to it.

```
const obj = { name: 'John' };
Object.preventExtensions(obj);

obj.age = 30; // This will not work, as the object is not extensible
console.log(obj.age); // undefined
```

What are JavaScript object getters and setters for?

JavaScript object getters and setters are used to control access to an object's properties. They provide a way to encapsulate the implementation details of a property and define custom behavior when getting or setting its value.

Getters and setters are defined using the `get` and `set` keywords, respectively, followed by a function that is executed when the property is accessed or assigned a new value.

Here's a code example demonstrating the use of getters and setters:

```
const person = {
  _name: 'John Doe', // Private property

  get name() {
    // Getter
    return this._name;
  },
  set name(newName) {
    // Setter
    if (newName.trim().length > 0) {
      this._name = newName;
    } else {
      console.log('Invalid name');
    }
  },
};

// Accessing the name property using the getter
console.log(person.name); // Output: 'John Doe'

// Setting the name property using the setter
person.name = 'Jane Smith'; // Setter is called
console.log(person.name); // Output: 'Jane Smith'

person.name = '';
// Setter is called, but the value is not set due to validation
console.log(person.name); // Output: 'Jane Smith'
```

What are JavaScript object property flags and descriptors?

In JavaScript, property flags and descriptors manage the behavior and attributes of object properties.

Property flags

Property flags are used to specify the behavior of a property on an object. Here are the available flags:

- `writable` : Specifies whether the property can be written to. Defaults to `true` .
- `enumerable` : Specifies whether the property is enumerable. Defaults to `true` .
- `configurable` : Specifies whether the property can be deleted or its attributes changed. Default is `true` .

Property descriptors

These provide detailed information about an object's property, including its value and flags. They are retrieved using `Object.getOwnPropertyDescriptor()` and set using `Object.defineProperty()` .

The use cases of property descriptors are as follows:

- Making a property non-writable by setting `writable: false` to ensure data consistency.
- Hiding a property from enumeration by setting `enumerable: false` .
- Preventing property deletion and modification by setting `configurable: false` .

- Freezing or sealing objects to prevent modifications globally.

How do you reliably determine whether an object is empty?

To reliably determine whether an object is empty, you can use `Object.keys()` to check if the object has any enumerable properties. If the length of the array returned by `Object.keys()` is zero, the object is empty.

```
const isEmpty = (obj) => Object.keys(obj).length === 0;

const obj = {};
console.log(isEmpty(obj)); // true
```

What is the event loop in JavaScript runtimes?

The event loop is a concept within the JavaScript runtime environment regarding how asynchronous operations are executed within JavaScript engines. It works as such:

1. The JavaScript engine starts executing scripts, placing synchronous operations on the call stack.
2. When an asynchronous operation is encountered (e.g., `setTimeout()`, HTTP request), it is offloaded to the respective Web API or Node.js API to handle the operation in the background.
3. Once the asynchronous operation completes, its callback function is placed in the respective queues – task queues (also known as macrotask queues / callback queues) or microtask queues. We will refer to "task queue" as "macrotask queue" from here on to better differentiate from the microtask queue.
4. The event loop continuously monitors the call stack and executes items on the call stack. If/when the call stack is empty:
   1. Microtask queue is processed. Microtasks include promise callbacks (`then`, `catch`, `finally`), `MutationObserver` callbacks, and calls to `queueMicrotask()`. The event loop takes the first callback from the microtask queue and pushes it to the call stack for execution. This repeats until the microtask queue is empty.
   2. Macrotask queue is processed. Macrotasks include web APIs like `setTimeout()`, HTTP requests, user interface event handlers like clicks, scrolls, etc. The event loop dequeues the first callback from the macrotask queue and pushes it onto the call stack for execution. However, after a macrotask queue callback is processed, the event loop does not proceed

with the next macrotask yet! The event loop first checks the microtask queue. Checking the microtask queue is necessary as microtasks have higher priority than macrotask queue callbacks. The macrotask queue callback that was just executed could have added more microtasks!

   1. If the microtask queue is non-empty, process them as per the previous step.
   2. If the microtask queue is empty, the next macrotask queue callback is processed. This repeats until the macrotask queue is empty.

5. This process continues indefinitely, allowing the JavaScript engine to handle both synchronous and asynchronous operations efficiently without blocking the call stack.

Explain the difference between synchronous and asynchronous functions in JavaScript

Synchronous functions are blocking while asynchronous functions are not. In synchronous functions, statements complete before the next statement is run. As a result, programs containing only synchronous code are evaluated exactly in order of the statements. The execution of the program is paused if one of the statements take a very long time.

```
function sum(a, b) {
  console.log('Inside sum function');
  return a + b;
}

const result = sum(2, 3);
// The program waits for sum() to complete before assigning the result
console.log('Result: ', result); // Output: 5
```

Asynchronous functions usually accept a callback as a parameter and execution continue on to the next line immediately after the asynchronous function is invoked. The callback is only invoked when the asynchronous operation is complete and the call stack is empty. Heavy duty operations such as loading data from a web server or querying a database should be done asynchronously so that the main thread can continue executing other operations instead of blocking until that long operation to complete (in the case of browsers, the UI will freeze).

```
function fetchData(callback) {
  setTimeout(() => {
    const data = { name: 'John', age: 30 };
    callback(data); // Calling the callback function with data
  }, 2000); // Simulating a 2-second delay
}

console.log('Fetching data...');

fetchData((data) => {
  console.log(data);
  // Output: { name: 'John', age: 30 } (after 2 seconds)
});

console.log('Call made to fetch data');
// This will print before the data is fetched
```

Explain the concept of a callback function in asynchronous operations

A callback function is a function passed as an argument to another function, which is then invoked inside the outer function to complete some kind of routine or action. In asynchronous operations, callbacks are used to handle tasks that take time to complete, such as network requests or file I/O, without blocking the execution of the rest of the code. For example:

```
function fetchData(callback) {
  setTimeout(() => {
    const data = { name: 'John', age: 30 };
    callback(data);
  }, 1000);
}

fetchData((data) => {
  console.log(data);
});
```

What are Promises and how do they work?

Promises in JavaScript are objects that represent the eventual completion (or failure) of an asynchronous operation and its resulting value. They have three states: `pending`, `fulfilled`, and `rejected`. You can handle the results of a promise using the `.then()` method for success and the `.catch()` method for errors.

```
let promise = new Promise((resolve, reject) => {
  // asynchronous operation
  const success = true;
  if (success) {
    resolve('Success!');
  } else {
    reject('Error!');
  }
});

promise
  .then((result) => {
    console.log(result); // 'Success!' (this will print)
  })
  .catch((error) => {
    console.error(error); // 'Error!'
  });
```

Explain the different states of a Promise

A `Promise` in JavaScript can be in one of three states: `pending`, `fulfilled`, or `rejected`. When a `Promise` is created, it starts in the `pending` state. If the operation completes successfully, the `Promise` transitions to the `fulfilled` state, and if it fails, it transitions to the `rejected` state. Here's a quick example:

```
let promise = new Promise((resolve, reject) => {
  // some asynchronous operation
  if (success) {
    resolve('Success!');
  } else {
    reject('Error!');
  }
});
```

What are the pros and cons of using Promises instead of callbacks in JavaScript?

Promises offer a cleaner alternative to callbacks, helping to avoid callback hell and making asynchronous code more readable. They facilitate writing sequential and parallel asynchronous operations with ease. However, using promises may introduce slightly more complex code.

What is the use of `Promise.all()`

`Promise.all()` is a method in JavaScript that takes an array of promises and returns a single promise. This returned promise resolves when all the input promises have resolved, or it rejects if any of the input promises reject. It is useful for running multiple asynchronous operations in parallel and waiting for all of them to complete.

```
const promise1 = Promise.resolve(3);
const promise2 = 42;
const promise3 = new Promise((resolve, reject) => {
  setTimeout(resolve, 100, 'foo');
});

Promise.all([promise1, promise2, promise3]).then((values) => {
  console.log(values); // [3, 42, 'foo']
});
```

How is `Promise.all()` different from `Promise.allSettled()`?

`Promise.all()` and `Promise.allSettled()` are both methods for handling multiple promises in JavaScript, but they behave differently. `Promise.all()` waits for all promises to resolve and fails fast if any promise rejects, returning a single rejected promise. `Promise.allSettled()`, on the other hand, waits for all promises to settle (either resolve or reject) and returns an array of objects describing the outcome of each promise.

What is async/await and how does it simplify asynchronous code?

`async/await` is a modern syntax in JavaScript that simplifies working with promises. By using the `async` keyword before a function, you can use the `await` keyword inside that function to pause execution until a promise is resolved. This makes asynchronous code look and behave more like synchronous code, making it easier to read and maintain.

```
async function fetchData() {
  try {
    const response = await fetch(
      'https://jsonplaceholder.typicode.com/posts/1',
    );
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error fetching data:', error);
  }
}
fetchData();
```

How do you handle errors in asynchronous operations?

To handle errors in asynchronous operations, you can use `try...catch` blocks with `async/await` syntax or `.catch()` method with Promises. For example, with `async/await`, you can wrap your code in a `try...catch` block to catch any errors:

```
async function fetchData() {
  try {
    // Invalid URl
    const response = await fetch('https://api.example.com/data');
    const data = await response.json();
    console.log(data);
  } catch (error) {
    console.error('Error fetching data:', error);
  }
}

fetchData(); // Error fetching data: ....
```

With Promises, you can use the `.catch()` method:

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

Explain the concept of a microtask queue

The microtask queue is a queue of tasks that need to be executed after the currently executing script and before any other task. Microtasks are typically used for tasks that need to be executed immediately after the current operation, such as promise callbacks. The microtask queue is processed before the macrotask queue, ensuring that microtasks are executed as soon as possible.

What is the difference between `setTimeout()`, `setImmediate()`, and `process.nextTick()`?

`setTimeout()` schedules a callback to run after a minimum delay. `setImmediate()` schedules a callback to run after the current event loop completes. `process.nextTick()` schedules a callback to run before the next event loop iteration begins.

```
setTimeout(() => console.log('setTimeout'), 0);
setImmediate(() => console.log('setImmediate'));
process.nextTick(() => console.log('nextTick'));
```

In this example, `process.nextTick()` will execute first, followed by either `setTimeout()` or `setImmediate()` depending on the environment.

Explain how prototypal inheritance works in JavaScript

Prototypical inheritance in JavaScript is a way for objects to inherit properties and methods from other objects. Every JavaScript object has a special hidden property called `[[Prototype]]` (commonly accessed via `__proto__` or using `Object.getPrototypeOf()`) that is a reference to another object, which is called the object's "prototype".

When a property is accessed on an object and if the property is not found on that object, the JavaScript engine looks at the object's `__proto__`, and the `__proto__`'s `__proto__` and so on, until it finds the property defined on one of the `__proto__`s or until it reaches the end of the prototype chain.

Here's an example of prototypal inheritance:

```
// Parent object constructor.
function Animal(name) {
  this.name = name;
}

// Add a method to the parent object's prototype.
Animal.prototype.makeSound = function () {
  console.log('The ' + this.constructor.name + ' makes a sound.');
};

// Child object constructor.
```

```
function Dog(name) {
  Animal.call(this, name); // Call the parent constructor.
}

// Set the child object's prototype to be the parent's prototype.
Object.setPrototypeOf(Dog.prototype, Animal.prototype);

// Add a method to the child object's prototype.
Dog.prototype.bark = function () {
  console.log('Woof!');
};

// Create a new instance of Dog.
const bolt = new Dog('Bolt');

// Call methods on the child object.
console.log(bolt.name); // "Bolt"
bolt.makeSound(); // "The Dog makes a sound."
bolt.bark(); // "Woof!"
```

Things to note are:

- `.makeSound` is not defined on `Dog`, so the JavaScript engine goes up the prototype chain and finds `.makeSound` on the inherited `Animal`.
- Using `Object.create()` to build the inheritance chain is no longer recommended. Use `Object.setPrototypeOf()` instead.

What is the prototype chain and how does it work?

The prototype chain is a mechanism in JavaScript that allows objects to inherit properties and methods from other objects. When you try to access a property on an object, JavaScript will first look for the property on the object itself. If it doesn't find it, it will look at the object's prototype, and then the prototype's prototype, and so on, until it either finds the property or reaches the end of the chain, which is `null`.

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

Person.prototype.greet = function () {
  console.log(`Hello, my name is ${this.name}`);
};
```

```
const alice = new Person('Alice');
alice.greet(); // "Hello, my name is Alice"
```

In this example, `alice` inherits the `greet` method from `Person.prototype`.

Explain the difference between classical inheritance and prototypal inheritance

Classical inheritance is a model where classes inherit from other classes, typically seen in languages like Java and C++. Prototypal inheritance, used in JavaScript, involves objects inheriting directly from other objects. In classical inheritance, you define a class and create instances from it. In prototypal inheritance, you create an object and use it as a prototype for other objects.

Explain the concept of inheritance in ES2015 classes

Inheritance in ES2015 classes allows one class to extend another, enabling the child class to inherit properties and methods from the parent class. This is done using the `extends` keyword. The `super` keyword is used to call the constructor and methods of the parent class. Here's a quick example:

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

  speak() {
    console.log(`${this.name} makes a noise.`);
  }
}

class Dog extends Animal {
  constructor(name, breed) {
    super(name);
    this.breed = breed;
  }

  speak() {
    console.log(`${this.name} barks.`);
  }
}

const dog = new Dog('Rex', 'German Shepherd');
dog.speak(); // Rex barks.
```

What is the purpose of the `new` keyword?

The `new` keyword in JavaScript is used to create an instance of a user-defined object type or one of the built-in object types that has a constructor function. When you use `new`, it does four things: it creates a new object, sets the prototype, binds `this` to the new object, and returns the new object.

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

const person1 = new Person('Alice');
console.log(person1.name); // Alice
```

How do you create a constructor function?

To create a constructor function in JavaScript, define a regular function with a capitalized name to indicate it's a constructor. Use the `this` keyword to set properties and methods. When creating an instance, use the `new` keyword.

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

const john = new Person('John', 30);
console.log(john.age); // 30
```

What are the differences between JavaScript ES2015 classes and ES5 function constructors?

ES2015 introduces a new way of creating classes, which provides a more intuitive and concise way to define and work with objects and inheritance compared to the ES5 function constructor syntax. Here's an example of each:

```
// ES5 function constructor
function Person(name) {
  this.name = name;
}

// ES2015 Class
class Person {
  constructor(name) {
    this.name = name;
  }
}
```

For simple constructors, they look pretty similar. The main difference in the constructor comes when using inheritance. If we want to create a `Student` class that subclasses `Person` and add a `studentId` field, this is what we have to do.

```
// ES5 inheritance
// Superclass
function Person1(name) {
  this.name = name;
}

// Subclass
function Student1(name, studentId) {
  // Call constructor of superclass
  // to initialize superclass-derived members.
  Person1.call(this, name);

  // Initialize subclass's own members.
  this.studentId = studentId;
}
Student1.prototype = Object.create(Person1.prototype);
Student1.prototype.constructor = Student1;

const student1 = new Student1('John', 1234);
console.log(student1.name, student1.studentId); // "John" 1234

// ES2015 inheritance
// Superclass
class Person2 {
  constructor(name) {
    this.name = name;
  }
}
```

```
// Subclass
class Student2 extends Person2 {
  constructor(name, studentId) {
    super(name);
    this.studentId = studentId;
  }
}

const student2 = new Student2('Alice', 5678);
console.log(student2.name, student2.studentId); // "Alice" 5678
```

It's much more verbose to use inheritance in ES5 and the ES2015 version is easier to understand and remember.

Comparison of ES5 function constructors vs ES2015 classes

Feature	ES5 Function Constructor	ES2015 Class
Syntax	Uses function constructors and prototypes	Uses `class` keyword
Constructor	Function with properties assigned using `this`	`constructor` method inside the class
Method Definition	Defined on the prototype	Defined inside the class body
Static Methods	Added directly to the constructor function	Defined using the `static` keyword
Inheritance	Uses `Object.create()` and manually sets prototype chain	Uses `extends` keyword and `super` function
Readability	Less intuitive and more verbose	More concise and intuitive

What advantage is there for using the JavaScript arrow syntax for a method in a constructor?

The main advantage of using an arrow function as a method inside a constructor is that the value of `this` gets set at the time of the function creation and can't change after that. When the constructor is used to create a new object, `this` will always refer to that object.

For example, let's say we have a `Person` constructor that takes a first name as an argument has two methods to `console.log()` that name, one as a regular function and one as an arrow function:

```
const Person = function (name) {
  this.firstName = name;
  this.sayName1 = function () {
    console.log(this.firstName);
  };
  this.sayName2 = () => {
    console.log(this.firstName);
  };
};

const john = new Person('John');
const dave = new Person('Dave');

john.sayName1(); // John
john.sayName2(); // John

// The regular function can have its `this` value changed,
// but the arrow function cannot
john.sayName1.call(dave);
// Dave (because `this` is now the dave object)
john.sayName2.call(dave); // John

john.sayName1.apply(dave);
// Dave (because `this` is now the dave object)
john.sayName2.apply(dave); // John

john.sayName1.bind(dave)();
// Dave (because `this` is now the dave object)
john.sayName2.bind(dave)(); // John

const sayNameFromWindow1 = john.sayName1;
sayNameFromWindow1();
// undefined (because `this` is now the window object)

const sayNameFromWindow2 = john.sayName2;
sayNameFromWindow2(); // John
```

The main takeaway here is that `this` can be changed for a normal function, but `this` always stays the same for an arrow function. So even if you are passing around your arrow function to different parts of your application, you wouldn't have to worry about the value of `this` changing.

Why might you want to create static class members in JavaScript?

Static class members (properties/methods) has a `static` keyword prepended. Such members cannot be directly accessed on instances of the class. Instead, they're accessed on the class itself.

```
class Car {
  static noOfWheels = 4;
  static compare() {
    return 'Static method has been called.';
  }
}

console.log(Car.noOfWheels); // 4
```

Static members are useful under the following scenarios:

- **Namespace organization**: Static properties can be used to define constants or configuration values that are specific to a class. This helps organize related data within the class namespace and prevents naming conflicts with other variables. Examples include `Math.PI`, `Math.SQRT2`.
- **Helper functions**: Static methods can be used as helper functions that operate on the class itself or its instances. This can improve code readability and maintainability by separating utility logic from the core functionality of the class. Examples of frequently used static methods include `Object.assign()`, `Math.max()`.
- **Singleton pattern**: In some rare cases, static properties and methods can be used to implement a singleton pattern, where only one instance of a class ever exists. However, this pattern can be tricky to manage and is generally discouraged in favor of more modern dependency injection techniques.

What is a closure in JavaScript, and how/why would you use one?

Closure is when a function is able to remember and access its lexical scope even when that function is executing outside its lexical scope

In simple terms, functions have access to variables that were in their scope at the time of their creation. This is what we call the function's lexical scope. A closure is a function that retains access to these variables even after the outer function has finished executing. This is like the function has a memory of its original environment.

```
function outerFunction() {
  const outerVar = 'I am outside of innerFunction';

  function innerFunction() {
    console.log(outerVar);
    // `innerFunction` can still access `outerVar`.
  }

  return innerFunction;
}

const inner = outerFunction();
// `inner` now holds a reference to `innerFunction`.

inner();
// "I am outside of innerFunction"
//  Even though `outerFunction` has completed execution,
// `inner` still has access to variables
//  defined inside `outerFunction`.```
```

Key points to remember:

- Closure occurs when an inner function has access to variables in its outer (lexical) scope, even when the outer function has finished executing.
- Closure allows a function to **remember** the environment in which it was created, even if that environment is no longer present.
- Closures are used extensively in JavaScript, such as in callbacks, event handlers, and asynchronous functions.

Explain the concept of lexical scoping

Lexical scoping means that the scope of a variable is determined by its location within the source code, and nested functions have access to variables declared in their outer scope. For example:

```
function outerFunction() {
  let outerVariable = 'I am outside!';
```

```
  function innerFunction() {
    console.log(outerVariable); // 'I am outside!'
  }

  innerFunction();
}

outerFunction();
```

In this example, `innerFunction` can access `outerVariable` because of lexical scoping.

Explain the concept of scope in JavaScript

In JavaScript, scope determines the accessibility of variables and functions at different parts of the code. There are three main types of scope: global scope, function scope, and block scope. Global scope means the variable is accessible everywhere in the code. Function scope means the variable is accessible only within the function it is declared. Block scope, introduced with ES6, means the variable is accessible only within the block (e.g., within curly braces `{}`) it is declared.

```
var globalVar = 'I am a global var';

function myFunction() {
  var functionVar = 'I am a function-scoped var';

  if (true) {
    let blockVar = 'I am a block-scoped var';

    console.log('Inside block:');
    console.log(globalVar); // Accessible
    console.log(functionVar); // Accessible
    console.log(blockVar); // Accessible
  }

  console.log('Inside function:');
  console.log(globalVar); // Accessible
  console.log(functionVar); // Accessible
  // console.log(blockVar); // Uncaught ReferenceError
}

myFunction();
```

```
console.log('In global scope:');
console.log(globalVar); // Accessible
// console.log(functionVar); // Uncaught ReferenceError
// console.log(blockVar); // Uncaught ReferenceError
```

How can closures be used to create private variables?

Closures in JavaScript can be used to create private variables by defining a function within another function. The inner function has access to the outer function's variables, but those variables are not accessible from outside the outer function. This allows you to encapsulate and protect the variables from being accessed or modified directly.

```
function createCounter() {
  let count = 0; // private variable

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

const counter = createCounter();
console.log(counter.increment()); // 1
console.log(counter.getCount()); // 1
console.log(counter.count); // undefined
```

What are the potential pitfalls of using closures?

Closures can lead to memory leaks if not managed properly, especially when they capture variables that are no longer needed. They can also make debugging more difficult due to the complexity of the scope chain. Additionally, closures can cause performance issues if they are overused or used inappropriately, as they keep references to variables in their scope, which can prevent garbage collection.

Explain the difference between global scope, function scope, and block scope

Global scope means variables are accessible from anywhere in the code. Function scope means variables are accessible only within the function they are declared in. Block scope means variables are accessible only within the block (e.g., within `{}`) they are declared in.

```
var globalVar = "I'm global"; // Global scope

function myFunction() {
  var functionVar = "I'm in a function"; // Function scope
  if (true) {
    let blockVar = "I'm in a block"; // Block scope
    console.log(blockVar); // Accessible here
  }
  // console.log(blockVar);
  // Uncaught ReferenceError: blockVar is not defined
}
// console.log(functionVar);
// Uncaught ReferenceError: functionVar is not defined
myFunction();
```

Explain how `this` works in JavaScript

There's no simple explanation for `this`; it is one of the most confusing concepts in JavaScript because it's behavior differs from many other programming languages. The one-liner explanation of the `this` keyword is that it is a dynamic reference to the context in which a function is executed.

A longer explanation is that `this` follows these rules:

1. If the `new` keyword is used when calling the function, meaning the function was used as a function constructor, the `this` inside the function is the newly-created object instance.
2. If `this` is used in a `class` `constructor`, the `this` inside the `constructor` is the newly-created object instance.

3. If `apply()`, `call()`, or `bind()` is used to call/create a function, `this` inside the function is the object that is passed in as the argument.
4. If a function is called as a method (e.g. `obj.method()`)— `this` is the object that the function is a property of.
5. If a function is invoked as a free function invocation, meaning it was invoked without any of the conditions present above, `this` is the global object. In the browser, the global object is the `window` object. If in strict mode (`'use strict';`), `this` will be `undefined` instead of the global object.
6. If multiple of the above rules apply, the rule that is higher wins and will set the `this` value.
7. If the function is an ES2015 arrow function, it ignores all the rules above and receives the `this` value of its surrounding scope at the time it is created.

Explain `Function.prototype.bind` in JavaScript

`Function.prototype.bind` is a method in JavaScript that allows you to create a new function with a specific `this` value and optional initial arguments. It's primary purpose is to:

- **Binding `this` value to preserve context**: The primary purpose of `bind` is to bind the `this` value of a function to a specific object. When you call `func.bind(thisArg)`, it creates a new function with the same body as `func`, but with `this` permanently bound to `thisArg`.
- **Partial application of arguments**: `bind` also allows you to pre-specify arguments for the new function. Any arguments passed to `bind` after `thisArg` will be prepended to the arguments list when the new function is called.
- **Method borrowing**: `bind` allows you to borrow methods from one object and apply them to another object, even if they were not originally designed to work with that object.

The `bind` method is particularly useful in scenarios where you need to ensure that a function is called with a specific `this` context, such as in event handlers, callbacks, or method borrowing.

Explain the different ways the `this` keyword can be bound

The `this` keyword in JavaScript can be bound in several ways:

- Default binding: In non-strict mode, `this` refers to the global object (window in browsers). In strict mode, `this` is `undefined`.
- Implicit binding: When a function is called as a method of an object, `this` refers to the object.
- Explicit binding: Using `call`, `apply`, or `bind` methods to explicitly set `this`.
- New binding: When a function is used as a constructor with the `new` keyword, `this` refers to the newly created object.
- Arrow functions: Arrow functions do not have their own `this` and inherit `this` from the surrounding lexical context.

What are the common pitfalls of using the `this` keyword?

The `this` keyword in JavaScript can be tricky because its value depends on how a function is called. Common pitfalls include losing the context of `this` when passing methods as callbacks, using `this` in nested functions, and misunderstanding `this` in arrow functions. To avoid these issues, you can use `.bind()`, arrow functions, or store the context in a variable.

Explain the concept of `this` binding in event handlers

In JavaScript, the `this` keyword refers to the object that is currently executing the code. In event handlers, `this` typically refers to the element that triggered the event. However, the value of `this` can change depending on how the event handler is defined and called. To ensure `this` refers to the desired object, you can use methods like `bind()`, arrow functions, or assign the context explicitly.

What is the DOM and how is it structured?

The DOM, or Document Object Model, is a programming interface for web documents. It represents the page so that programs can change the document structure, style, and content. The DOM is structured as a tree of objects, where each node represents part of the document, such as elements, attributes, and text.

What's the difference between an "attribute" and a "property" in the DOM?

Attributes are defined in the HTML and provide initial values for properties. Properties are part of the DOM and represent the current state of an element. For example, the `value` attribute of an `<input>` element sets its initial value, while the `value` property reflects the current value as the user interacts with it.

Explain the difference between `document.querySelector()` and `document.getElementById()`

`document.querySelector()` and `document.getElementById()` are both methods used to select elements from the DOM, but they have key differences. `document.querySelector()` can select any element using a CSS selector and returns the first match, while `document.getElementById()` selects an element by its ID and returns the element with that specific ID.

```
// Using document.querySelector()
const element = document.querySelector('.my-class');

// Using document.getElementById()
const elementById = document.getElementById('my-id');
```

How do you add, remove, and modify HTML elements using JavaScript?

To add, remove, and modify HTML elements using JavaScript, you can use methods like `createElement`, `appendChild`, `removeChild`, and properties like `innerHTML` and `textContent`. For example, to add an element, you can create it using `document.createElement` and then append it to a parent element using `appendChild`. To remove an element, you can use `removeChild` on its parent. To modify an element, you can change its `innerHTML` or `textContent`.

```
// Adding an element
const newElement = document.createElement('div');
newElement.textContent = 'Hello, World!';
document.body.appendChild(newElement);

// Removing an element
const elementToRemove = document.getElementById('elementId');
elementToRemove.parentNode.removeChild(elementToRemove);

// Modifying an element
```

```
const elementToModify = document.getElementById('elementId');
elementToModify.innerHTML = 'New Content';
```

What are event listeners and how are they used?

Event listeners are functions that wait for specific events to occur on elements, such as clicks or key presses. They are used to execute code in response to these events. You can add an event listener to an element using the `addEventListener` method. For example:

```
document.getElementById('myButton').addEventListener('click',
function () {
  alert('Button was clicked!');
});
```

Explain the event phases in a browser

In a browser, events go through three phases: capturing, target, and bubbling. During the capturing phase, the event travels from the root to the target element. In the target phase, the event reaches the target element. Finally, in the bubbling phase, the event travels back up from the target element to the root. You can control event handling using `addEventListener` with the `capture` option.

Describe event bubbling in JavaScript and browsers

Event bubbling is a DOM event propagation mechanism where an event (e.g. a click), starts at the target element and bubbles up to the root of the document. This allows ancestor elements to also respond to the event.

Event bubbling is essential for event delegation, where a single event handler manages events for multiple child elements, enhancing performance and code simplicity. While convenient, failing to manage event propagation properly can lead to unintended behavior, such as multiple handlers firing for a single event.

Describe event capturing in JavaScript and browsers

Event capturing is a lesser-used counterpart to event bubbling in the DOM event propagation mechanism. It follows the opposite order, where an event triggers first on the ancestor element and then travels down to the target element.

Event capturing is rarely used as compared to event bubbling, but it can be used in specific scenarios where you need to intercept events at a higher level before they reach the target element. It is disabled by default but can be enabled through an option on `addEventListener()`.

Explain event delegation in JavaScript

Event delegation is a technique in JavaScript where a single event listener is attached to a parent element instead of attaching event listeners to multiple child elements. When an event occurs on a child element, the event bubbles up the DOM tree, and the parent element's event listener handles the event based on the target element.

Event delegation provides the following benefits:

- **Improved performance**: Attaching a single event listener is more efficient than attaching multiple event listeners to individual elements, especially for large or dynamic lists. This reduces memory usage and improves overall performance.
- **Simplified event handling**: With event delegation, you only need to write the event handling logic once in the parent element's event listener. This makes the code more maintainable and easier to update.
- **Dynamic element support**: Event delegation automatically handles events for dynamically added or removed elements within the parent element. There's no need to manually attach or remove event listeners when the DOM structure changes

However, do note that:

- It is important to identify the target element that triggered the event.
- Not all events can be delegated because they are not bubbled. Non-bubbling events include: `focus`, `blur`, `scroll`, `mouseenter`, `mouseleave`, `resize`, etc.

How do you prevent the default behavior of an event?

To prevent the default behavior of an event in JavaScript, you can use the `preventDefault` method on the event object. For example, if you want to prevent a form from submitting, you can do the following:

```
document.querySelector('form').addEventListener('submit',
function (event) {
  event.preventDefault();
});
```

This method stops the default action associated with the event from occurring.

What is the difference between `event.preventDefault()` and `event.stopPropagation()`?

`event.preventDefault()` is used to prevent the default action that belongs to the event, such as preventing a form from submitting. `event.stopPropagation()` is used to stop the event from bubbling up to parent elements, preventing any parent event handlers from being executed.

What is the difference between `mouseenter` and `mouseover` event in JavaScript and browsers?

The main difference lies in the bubbling behavior of `mouseenter` and `mouseover` events. `mouseenter` does not bubble while `mouseover` bubbles.

`mouseenter` events do not bubble. The `mouseenter` event is triggered only when the mouse pointer enters the element itself, not its descendants. If a parent element has child elements, and the mouse pointer enters child elements, the `mouseenter` event will not be triggered on the parent element again, it's only triggered once upon entry of parent element without regard for its contents. If both parent and child have `mouseenter` listeners attached and the mouse pointer moves from the parent element to the child element, `mouseenter` will only fire for the child.

`mouseover` events bubble up the DOM tree. The `mouseover` event is triggered when the mouse pointer enters the element or one of its descendants. If a parent element has child elements, and the mouse pointer enters child elements, the `mouseover` event will be triggered on the parent element again as well. If the parent element has multiple child elements, this can result in multiple event callbacks fired. If there are child elements, and the mouse pointer moves from the parent element to the child element, `mouseover` will fire for both the parent and the child.

Property	`mouseenter`	`mouseover`
Bubbling	No	Yes
Trigger	Only when entering itself	When entering itself and when entering descendants

What is the difference between `innerHTML` and `textContent`?

`innerHTML` and `textContent` are both properties used to get or set the content of an HTML element, but they serve different purposes. `innerHTML` returns or sets the HTML markup contained within the element, which means it can parse and render HTML tags. On the other hand, `textContent` returns or sets the text content of the element, ignoring any HTML tags and rendering them as plain text.

```
// Example of innerHTML
element.innerHTML = '<strong>Bold Text</strong>';
// Renders as bold text

// Example of textContent
element.textContent = '<strong>Bold Text</strong>';
// Renders as plain text: <strong>Bold Text</strong>
```

How do you manipulate CSS styles using JavaScript?

You can manipulate CSS styles using JavaScript by accessing the `style` property of an HTML element. For example, to change the background color of a `div` element with the id `myDiv`, you can use:

```
document.getElementById('myDiv').style.backgroundColor = 'blue';
```

You can also add, remove, or toggle CSS classes using the `classList` property:

```
document.getElementById('myDiv').classList.add('newClass');
document.getElementById('myDiv').classList.remove('oldClass');
document.getElementById('myDiv').classList.toggle('toggleClass');
```

Describe the difference between `<script>`, `<script async>` and `<script defer>`

All of these ways (`<script>`, `<script async>`, and `<script defer>`) are used to load and execute JavaScript files in an HTML document, but they differ in how the browser handles loading and execution of the script:

- `<script>` is the default way of including JavaScript. The browser blocks HTML parsing while the script is being downloaded and executed. The browser will not continue rendering the page until the script has finished executing.
- `<script async>` downloads the script asynchronously, in parallel with parsing the HTML. Executes the script as soon as it is available, potentially interrupting the HTML parsing. `<script async>` do not wait for each other and execute in no particular order.
- `<script defer>` downloads the script asynchronously, in parallel with parsing the HTML. However, the execution of the script is deferred until HTML parsing is complete, in the order they appear in the HTML.

Here's a table summarizing the 3 ways of loading `<script>`s in a HTML document.

Feature	`<script>`	`<script async>`	`<script defer>`
Parsing behavior	Blocks HTML parsing	Runs parallel to parsing	Runs parallel to parsing
Execution order	In order of appearance	Not guaranteed	In order of appearance
DOM dependency	No	No	Yes (waits for DOM)

What is the difference between the Window object and the Document object?

The `Window` object represents the browser window and provides methods to control it, such as opening new windows or accessing the browser history. The `Document` object represents the content of the web page loaded in the window and provides methods to manipulate the DOM, such as selecting elements or modifying their content.

Describe the difference between a cookie, `sessionStorage` and `localStorage` in browsers

All of the following are mechanisms of storing data on the client, the user's browser in this case. `localStorage` and `sessionStorage` both implement the Web Storage API interface.

- **Cookies**: Suitable for server-client communication, small storage capacity, can be persistent or session-based, domain-specific. Sent to the server on every request.
- **`localStorage`**: Suitable for long-term storage, data persists even after the browser is closed, accessible across all tabs and windows of the same origin, highest storage capacity among the three.
- **`sessionStorage`**: Suitable for temporary data within a single page session, data is cleared when the tab or window is closed, has a higher storage capacity compared to cookies.

Here's a table summarizing the 3 client storage mechanisms.

Property	Cookie	`localStorage`	`sessionStorage`
Initiator	Client or server. Server can use `Set-Cookie` header	Client	Client
Lifespan	As specified	Until deleted	Until tab is closed
Persistent across browser sessions	If a future expiry date is set	Yes	No
Sent to server with every HTTP request	Yes, sent via `Cookie` header	No	No
Total capacity (per domain)	4kb	5MB	5MB
Access	Across windows/tabs	Across windows/tabs	Same tab
Security	JavaScript cannot access `HttpOnly` cookies	None	None

How do you make an HTTP request using the Fetch API?

To make an HTTP request using the Fetch API, you can use the `fetch` function, which returns a promise. You can handle the response using `.then()` and `.catch()` for error handling. Here's a basic example of a GET request:

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((error) => console.error('Error:', error));
```

For a POST request, you can pass an options object as the second argument to `fetch`:

```
fetch('https://jsonplaceholder.typicode.com/posts', {
  method: 'POST',
  body: JSON.stringify({
    title: 'foo',
    body: 'bar',
    userId: 1,
  }),
  headers: {
    'Content-Type': 'application/json; charset=UTF-8',
  },
})
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((error) => console.error('Error:', error));
```

What are the different ways to make an API call in JavaScript?

In JavaScript, you can make API calls using several methods. The most common ones are `XMLHttpRequest`, `fetch`, and third-party libraries like `Axios`. `XMLHttpRequest` is the traditional way but is more verbose. `fetch` is modern and returns promises, making it easier to work with. `Axios` is a popular third-party library that simplifies API calls and provides additional features.

Explain AJAX in as much detail as possible

AJAX (Asynchronous JavaScript and XML) facilitates asynchronous communication between the client and server, enabling dynamic updates to web pages without reloading. It uses techniques like `XMLHttpRequest` or the `fetch()` API to send and receive data in the background. In modern web applications, the `fetch()` API is more commonly used to implement AJAX.

Using `XMLHttpRequest`

```
let xhr = new XMLHttpRequest();
xhr.onreadystatechange = function () {
  if (xhr.readyState === XMLHttpRequest.DONE) {
    if (xhr.status === 200) {
      console.log(xhr.responseText);
    } else {
      console.error('Request failed: ' + xhr.status);
    }
  }
};
xhr.open('GET', 'https://jsonplaceholder.typicode.com/todos/1', true);
xhr.send();
```

Using `fetch()`

```
fetch('https://jsonplaceholder.typicode.com/todos/1')
  .then((response) => {
    if (!response.ok) {
      throw new Error('Network response was not ok');
    }
    return response.json();
  })
  .then((data) => console.log(data))
  .catch((error) => console.error('Fetch error:', error));
```

What are the advantages and disadvantages of using AJAX?

AJAX (Asynchronous JavaScript and XML) is a technique in JavaScript that allows web pages to send and retrieve data asynchronously from servers without refreshing or reloading the entire page.

Advantages

- **Smoother user experience**: Updates happen without full page reloads, like in mail and chat applications.

- **Lighter server Load**: Only necessary data is fetched via AJAX, reducing server load and improving perceived performance of webpages.
- **Maintains client state**: User interactions and any client states are persisted within the page.

Disadvantages

- **Reliance on JavaScript**: If disabled, Ajax functionality breaks.
- **Bookmarking issues**: Dynamic content makes bookmarking specific page states difficult.
- **SEO Challenges**: Search engines may struggle to index dynamic content.
- **Performance Concerns**: Processing Ajax data on low-end devices can be slow.

What are the differences between `XMLHttpRequest` and `fetch()` in JavaScript and browsers?

`XMLHttpRequest` (XHR) and `fetch()` API are both used for asynchronous HTTP requests in JavaScript (AJAX). `fetch()` offers a cleaner syntax, promise-based approach, and more modern feature set compared to XHR. However, there are some differences:

- `XMLHttpRequest` event callbacks, while `fetch()` utilizes promise chaining.
- `fetch()` provides more flexibility in headers and request bodies.
- `fetch()` support cleaner error handling with `catch()`.
- Handling caching with `XMLHttpRequest` is difficult but caching is supported by `fetch()` by default in the `options.cache` object (`cache` value of second parameter) to `fetch()` or `Request()`.
- `fetch()` requires an `AbortController` for cancelation, while for `XMLHttpRequest`, it provides `abort()` property.
- `XMLHttpRequest` has good support for progress tracking, which `fetch()` lacks.
- `XMLHttpRequest` is only available in the browser and not natively supported in Node.js environments. On the other hand `fetch()` is part of the JavaScript language and is supported on all modern JavaScript runtimes.

These days `fetch()` is preferred for its cleaner syntax and modern features.

How do you abort a web request using `AbortController` in JavaScript?

`AbortController` is used to cancel ongoing asynchronous operations like fetch requests.

```
const controller = new AbortController();
const signal = controller.signal;

fetch('https://jsonplaceholder.typicode.com/todos/1', { signal })
  .then((response) => {
    // Handle response
  })
  .catch((error) => {
    if (error.name === 'AbortError') {
      console.log('Request aborted');
    } else {
      console.error('Error:', error);
    }
  });

// Call abort() to abort the request
controller.abort();
```

Aborting web requests is useful for:

- Canceling requests based on user actions.
- Prioritizing the latest requests in scenarios with multiple simultaneous requests.
- Canceling requests that are no longer needed, e.g. after the user has navigated away from the page.

Explain how JSONP works (and how it's not really Ajax)

JSONP (JSON with Padding) is a technique used to overcome the same-origin policy in web browsers, allowing you to request data from a server in a different domain. It works by dynamically creating a `<script>` tag and setting its `src` attribute to the URL of the data source. The server responds with a script that calls a predefined callback function with the data as its argument. Unlike Ajax, JSONP does not use the XMLHttpRequest object and is limited to GET requests.

What are workers in JavaScript used for?

Workers in JavaScript are background threads that allow you to run scripts in parallel with the main execution thread, without blocking or interfering with the user interface.

Their key features include:

- **Parallel processing**: Workers run in a separate thread from the main thread, allowing your web page to remain responsive to user interactions while the worker performs its tasks. It's useful for moving CPU-intensive work off the main thread and be free from JavaScript's single-threaded nature.
- **Communication**: Uses `postMessage()` and `onmessage` / `'message'` event for messaging.
- **Access to web APIs**: Workers have access to various Web APIs, including `fetch()`, IndexedDB, and Web Storage, allowing them to perform tasks like data fetching and persisting data independently.
- **No DOM access**: Workers cannot directly manipulate the DOM, thus cannot interact with the UI, ensuring they don't accidentally interfere with the main thread's operation.

There are three main types of workers in JavaScript:

- **Web workers / Dedicated workers**
  - Run scripts in background threads, separate from the main UI thread.
  - Useful for CPU-intensive tasks like data processing, calculations, etc.
  - Cannot directly access or manipulate the DOM.
- **Service workers**
  - Act as network proxies, handling requests between the app and network.
  - Enable offline functionality, caching, and push notifications.
  - Runs independently of the web page, even when it's closed.
- **Shared workers**
  - Can be shared by multiple scripts running in different windows or frames, as long as they're in the same domain.
  - Scripts communicate with the shared worker by sending and receiving messages.
  - Useful for coordinating tasks across different parts of a web page.

Explain the concept of the Web Socket API

The WebSocket API provides a way to open a persistent connection between a client and a server, allowing for real-time, two-way communication. Unlike HTTP, which is request-response based, WebSocket enables full-duplex communication, meaning both the client and server can send and receive messages independently. This is particularly useful for applications like chat apps, live updates, and online gaming.

The following example uses Postman's WebSocket echo service to demonstrate how web sockets work.

```
// Postman's echo server that will echo back messages you send
const socket = new WebSocket('wss://ws.postman-echo.com/raw');

// Event listener for when the connection is open
socket.addEventListener('open', function (event) {
  socket.send('Hello Server!');
  // Sends the message to the Postman WebSocket server
});

// Event listener for when a message is received from the server
socket.addEventListener('message', function (event) {
  console.log('Message from server ', event.data);
});
```

What are JavaScript polyfills for?

Polyfills in JavaScript are pieces of code that provide modern functionality to older browsers that lack native support for those features. They bridge the gap between the JavaScript language features and APIs available in modern browsers and the limited capabilities of older browser versions.

They can be implemented manually or included through libraries and are often used in conjunction with feature detection.

Common use cases include:

- **New JavaScript Methods**: For example, `Array.prototype.includes()`, `Object.assign()`, etc.
- **New APIs**: Such as `fetch()`, `Promise`, `IntersectionObserver`, etc. Modern browsers support these now but for a long time they have to be polyfilled.

Libraries and services for polyfills:

- **`core-js`**: A modular standard library for JavaScript which includes polyfills for a wide range of ECMAScript features.

  ```
  import 'core-js/actual/array/flat-map';
  // With this, Array.prototype.flatMap is available to be used.
  ```

```
[1, 2].flatMap((it) => [it, it]); // => [1, 1, 2, 2]
```

- **Polyfill.io**: A service that provides polyfills based on the features and user agents specified in the request.

```
<script src="https://polyfill.io/v3/polyfill.min.js"></script>
```

How do you detect if JavaScript is disabled on a page?

To detect if JavaScript is disabled on a page, you can use the `<noscript>` HTML tag. This tag allows you to display content or messages to users who have JavaScript disabled in their browsers. For example, you can include a message within the `<noscript>` tag to inform users that JavaScript is required for the full functionality of the page.

```
<noscript>
  <p>
    JavaScript is disabled in your browser.
    Please enable JavaScript for the
    best experience.
  </p>
</noscript>
```

What is the `Intl` namespace object for?

The `Intl` namespace object in JavaScript is used for internationalization purposes. It provides language-sensitive string comparison, number formatting, and date and time formatting. For example, you can use `Intl.DateTimeFormat` to format dates according to a specific locale:

```
const date = new Date();
const formatter = new Intl.DateTimeFormat('en-US');
console.log(formatter.format(date));
```

How do you validate form elements using the Constraint Validation API?

The Constraint Validation API provides a way to validate form elements in HTML. You can use properties like `validity`, `validationMessage`, and methods like `checkValidity()` and `setCustomValidity()`. For example, to check if an input is valid, you can use:

```
const input = document.querySelector('input');
if (input.checkValidity()) {
  console.log('Input is valid');
} else {
  console.log(input.validationMessage);
}
```

How do you use `window.history` API?

The `window.history` API allows you to manipulate the browser's session history. You can use `history.pushState()` to add a new entry to the history stack, `history.replaceState()` to modify the current entry, and `history.back()`, `history.forward()`, and `history.go()` to navigate through the history. For example, `history.pushState({page: 1}, "title 1", "?page=1")` adds a new entry to the history.

How do `<iframe>` on a page communicate?

`<iframe>` elements on a page can communicate using the `postMessage` API. This allows for secure cross-origin communication between the parent page and the iframe. The `postMessage` method sends a message, and the `message` event listener receives it. Here's a simple example:

```
// In the parent page
const iframe = document.querySelector('iframe');
iframe.contentWindow.postMessage('Hello from parent', '*');

// In the iframe
window.addEventListener('message', (event) => {
  console.log(event.data); // 'Hello from parent'
});
```

Difference between document `load` event and document `DOMContentLoaded` event?

The `DOMContentLoaded` event fires when the initial HTML document has been completely loaded and parsed, without waiting for stylesheets, images, and subframes to finish loading. The `load` event, on the other hand, fires when the entire page, including all dependent resources such as stylesheets and images, has finished loading.

```
document.addEventListener('DOMContentLoaded', function () {
  console.log('DOM fully loaded and parsed');
});

window.addEventListener('load', function () {
  console.log('Page fully loaded');
});
```

How do you redirect to a new page in JavaScript?

To redirect to a new page in JavaScript, you can use the `window.location` object. The most common methods are `window.location.href` and `window.location.replace()`. For example:

```
// Using window.location.href
window.location.href = 'https://www.example.com';

// Using window.location.replace()
window.location.replace('https://www.example.com');
```

How do you get the query string values of the current page in JavaScript?

To get the query string values of the current page in JavaScript, you can use the `URLSearchParams` object. First, create a `URLSearchParams` instance with `window.location.search`, then use the `get` method to retrieve specific query parameters. For example:

```
const params = new URLSearchParams(window.location.search);
const value = params.get('language');
console.log(value);
```

What are server-sent events?

Server-sent events (SSE) is a standard that allows a web page to receive automatic updates from a server via an HTTP connection. Server-sent events are used with `EventSource` instances that opens a connection with a server and allows client to receive events from the server. Connections created by server-sent events are persistent (similar to the `WebSocket` s), however there are a few differences:

Property	`WebSocket`	`EventSource`
Direction	Bi-directional – both client and server can exchange messages	Unidirectional – only server sends data
Data type	Binary and text data	Only text
Protocol	WebSocket protocol (`ws://`)	Regular HTTP (`http://`)

Creating an event source

```
const eventSource = new EventSource('/sse-stream');
```

Listening for events

```
// Fired when the connection is established.
eventSource.addEventListener('open', () => {
  console.log('Connection opened');
});

// Fired when a message is received from the server.
eventSource.addEventListener('message', (event) => {
  console.log('Received message:', event.data);
});

// Fired when an error occurs.
eventSource.addEventListener('error', (error) => {
  console.error('Error occurred:', error);
});
```

Sending events from server

```
const express = require('express');
const app = express();

app.get('/sse-stream', (req, res) => {
  // `Content-Type` need to be set to `text/event-stream`.
  res.setHeader('Content-Type', 'text/event-stream');
  res.setHeader('Cache-Control', 'no-cache');
  res.setHeader('Connection', 'keep-alive');

  // Each message should be prefixed with data.
  const sendEvent = (data) => res.write(`data: ${data}\n\n`);

  sendEvent('Hello from server');

  const intervalId = setInterval(
      () => sendEvent(new Date().toString()),
  1000);

  res.on('close', () => {
    console.log('Client closed connection');
    clearInterval(intervalId);
  });
});

app.listen(3000, () => console.log('Server started on port 3000'));
```

In this example, the server sends a "Hello from server" message initially, and then sends the current date every second. The connection is kept alive until the client closes it

What are Progressive Web Applications (PWAs)?

Progressive Web Applications (PWAs) are web applications that use modern web capabilities to deliver an app-like experience to users. They are reliable, fast, and engaging. PWAs can work offline, send push notifications, and be installed on a user's home screen. They leverage technologies like service workers, web app manifests, and HTTPS to provide these features.

What are modules and why are they useful?

Modules are reusable pieces of code that can be imported and exported between different files in a project. They help in organizing code, making it more maintainable and scalable. By using modules, you can avoid global namespace pollution and manage dependencies more effectively. In JavaScript, you can use `import` and `export` statements to work with modules.

```
// myModule.js
export const myFunction = () => {
  console.log('Hello, World!');
};

// main.js
import { myFunction } from './myModule.js';
myFunction(); // Outputs: Hello, World!
```

Explain the differences between CommonJS modules and ES modules in JavaScript

In JavaScript, modules are reusable pieces of code that encapsulate functionality, making it easier to manage, maintain, and structure your applications. Modules allow you to break down your code into smaller, manageable parts, each with its own scope.

CommonJS is an older module system that was initially designed for server-side JavaScript development with Node.js. It uses the `require()` function to load modules and the `module.exports` or `exports` object to define the exports of a module.

```
// my-module.js
const value = 42;
module.exports = { value };

// main.js
const myModule = require('./my-module.js');
console.log(myModule.value); // 42
```

ES Modules (ECMAScript Modules) are the standardized module system introduced in ES6 (ECMAScript 2015). They use the `import` and `export` statements to handle module dependencies.

```
// my-module.js
export const value = 42;

// main.js
import { value } from './my-module.js';
console.log(value); // 42
```

CommonJS vs ES modules

Feature	CommonJS	ES modules
Module Syntax	`require()` for importing `module.exports` for exporting	`import` for importing `export` for exporting
Environment	Primarily used in Node.js for server-side development	Designed for both browser and server-side JavaScript (Node.js)
Loading	Synchronous loading of modules	Asynchronous loading of modules
Structure	Dynamic imports, can be conditionally called	Static imports/exports at the top level
File extensions	`.js` (default)	`.mjs` or `.js` (with `type: "module"` in `package.json`)
Browser support	Not natively supported in browsers	Natively supported in modern browsers
Optimization	Limited optimization due to dynamic nature	Allows for optimizations like tree-shaking due to static structure
Compatibility	Widely used in existing Node.js codebases and libraries	Newer standard, but gaining adoption in modern projects

How do you import and export modules in JavaScript?

In JavaScript, you can import and export modules using the `import` and `export` statements. To export a module, you can use `export` before a function, variable, or class, or use `export default` for a single default export. To import a module, you use the `import` statement followed by the name of the exported module and the path to the module file.

```
// Exporting a module
export const myFunction = () => {
  /* ... */
};
```

```
export default myFunction;

// Importing a module
import { myFunction } from './myModule';
import myFunction from './myModule';
```

What are the benefits of using a module bundler?

Using a module bundler like Webpack, Rollup, or Parcel helps manage dependencies, optimize performance, and improve the development workflow. It combines multiple JavaScript files into a single file or a few files, which reduces the number of HTTP requests and can include features like code splitting, tree shaking, and hot module replacement.

Explain the concept of tree shaking in module bundling

Tree shaking is a technique used in module bundling to eliminate dead code, which is code that is never used or executed. This helps to reduce the final bundle size and improve application performance. It works by analyzing the dependency graph of the code and removing any unused exports. Tools like Webpack and Rollup support tree shaking when using ES6 module syntax (`import` and `export`).

What are the metadata fields of a module?

Metadata fields of a module typically include information such as the module’s name, version, description, author, license, and dependencies. These fields are often found in a `package.json` file in JavaScript projects. For example:

```
{
  "name": "my-module",
  "version": "1.0.0",
  "description": "A sample module",
  "author": "John Doe",
  "license": "MIT",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

What do you think of CommonJS vs ESM?

JavaScript has evolved its module systems. ESM (ECMAScript Modules) using `import` / `export` is the official standard, natively supported in modern browsers and Node.js, designed for both synchronous and asynchronous use cases. CommonJS (CJS) using `require` / `module.exports` was the original standard for Node.js, primarily synchronous, and remains prevalent in the Node ecosystem. AMD (Asynchronous Module Definition) using `define` / `require` was an early system designed for asynchronous loading in browsers but is now largely obsolete, replaced by ESM.

What are the different types of errors in JavaScript?

In JavaScript, there are three main types of errors: syntax errors, runtime errors, and logical errors. Syntax errors occur when the code violates the language's grammar rules, such as missing a parenthesis. Runtime errors happen during code execution, like trying to access a property of `undefined`. Logical errors are mistakes in the code's logic that lead to incorrect results but don't throw an error.

How do you handle errors using `try...catch` blocks?

To handle errors using `try...catch` blocks, you wrap the code that might throw an error inside a `try` block. If an error occurs, the control is transferred to the `catch` block where you can handle the error. Optionally, you can use a `finally` block to execute code regardless of whether an error occurred or not.

```
try {
  // Code that may throw an error
} catch (error) {
  // Handle the error
} finally {
  // Code that will run regardless of an error
}
```

What is the purpose of the `finally` block?

The `finally` block in JavaScript is used to execute code after a `try` and `catch` block, regardless of whether an error was thrown or caught. It ensures that certain cleanup or finalization code runs no matter what. For example:

```
try {
  // Code that may throw an error
} catch (error) {
  // Code to handle the error
} finally {
  // Code that will always run
}
```

How can you create custom error objects?

To create custom error objects in JavaScript, you can extend the built-in `Error` class. This allows you to add custom properties and methods to your error objects. Here's a quick example:

```
class CustomError extends Error {
  constructor(message) {
    super(message);
    this.name = 'CustomError';
  }
}

try {
  throw new CustomError('This is a custom error message');
} catch (error) {
  console.log(error.name); // CustomError
  console.log(error.message); // This is a custom error message
}
```

Explain the concept of error propagation in JavaScript

Error propagation in JavaScript refers to how errors are passed through the call stack. When an error occurs in a function, it can be caught and handled using `try...catch` blocks. If not caught, the error propagates up the call stack until it is either caught or causes the program to terminate. For example:

```
function a() {
  throw new Error('An error occurred');
}

function b() {
  a();
}

try {
  b();
} catch (e) {
  console.error(e.message); // Outputs: An error occurred
}
```

What is currying and how does it work?

Currying is a technique in functional programming where a function that takes multiple arguments is transformed into a series of functions that each take a single argument. This allows for partial application of functions. For example, a function `f(a, b, c)` can be curried into `f(a)(b)(c)`. Here's a simple example in JavaScript:

```
function add(a) {
  return function (b) {
    return function (c) {
      return a + b + c;
    };
  };
}

const addOne = add(1);
console.log(addOne); // function object

const addOneAndTwo = addOne(2);
console.log(addOneAndTwo); // function object

const result = addOneAndTwo(3);
console.log(result); // Output: 6
```

Explain the concept of partial application

Partial application is a technique in functional programming where a function is applied to some of its arguments, producing a new function that takes the remaining arguments. This allows you to create more specific functions from general ones. For example, if you have a function `add(a, b)`, you can partially apply it to create a new function `add5` that always adds 5 to its argument.

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

const add5 = add.bind(null, 5);
console.log(add5(10)); // Outputs 15
```

What are the benefits of using currying and partial application?

Currying transforms a function with multiple arguments into a sequence of functions, each taking a single argument. This allows for more flexible and reusable code. Partial application, on the other hand, allows you to fix a few arguments of a function and generate a new function. Both techniques help in creating more modular and maintainable code.

Provide some examples of how currying and partial application can be used

Currying transforms a function with multiple arguments into a sequence of functions, each taking a single argument. Partial application fixes a few arguments of a function, producing another function with a smaller number of arguments. For example, currying a function `add(a, b)` would look like `add(a)(b)`, while partial application of `add(2, b)` would fix the first argument to 2, resulting in a function that only needs the second argument.

Currying example:

```
const add = (a) => (b) => a + b;
const addTwo = add(2);
console.log(addTwo(3)); // 5
```

Partial application example:

```
const add = (a, b) => a + b;
const addTwo = add.bind(null, 2);
console.log(addTwo(3)); // 5
```

How do currying and partial application differ from each other?

Currying transforms a function with multiple arguments into a sequence of functions, each taking a single argument. For example, a function `f(a, b, c)` becomes `f(a)(b)(c)`. Partial application, on the other hand, fixes a few arguments of a function and produces another function with a smaller number of arguments. For example, if you partially apply `f(a, b, c)` with `a`, you get a new function `f'(b, c)`.

What are `Set`s and `Map`s and how are they used?

`Set`s and `Map`s are built-in JavaScript objects that help manage collections of data. A `Set` is a collection of unique values, while a `Map` is a collection of key-value pairs where keys can be of any type. `Set`s are useful for storing unique items, and `Map`s are useful for associating values with keys.

```
// Set example
let mySet = new Set([1, 2, 3, 3]);
// Set {1, 2, 3} (duplicate values are not added)
mySet.add(4);
console.log(mySet); // Set {1, 2, 3, 4}

// Map example
let myMap = new Map();
myMap.set('key1', 'value1');
myMap.set('key2', 'value2');
console.log(myMap.get('key1')); // 'value1'
```

What are the differences between `Map`/`Set` and `WeakMap`/`WeakSet` in JavaScript?

The primary difference between `Map`/`Set` and `WeakMap`/`WeakSet` in JavaScript lies in how they handle keys. Here's a breakdown:

`Map` vs. `WeakMap`

`Map`s allows any data type (strings, numbers, objects) as keys. The key-value pairs remain in memory as long as the `Map` object itself is referenced. Thus they are suitable for general-purpose key-value storage where you want to maintain references to both keys and values. Common use cases include storing user data, configuration settings, or relationships between objects.

`WeakMap`s only allows objects as keys. However, these object keys are held weakly. This means the garbage collector can remove them from memory even if the `WeakMap` itself still exists, as long as there are no other references to those objects. `WeakMap`s are ideal for scenarios where you want to associate data with objects without preventing those objects from being garbage collected. This can be useful for things like:

- Caching data based on objects without preventing garbage collection of the objects themselves.
- Storing private data associated with DOM nodes without affecting their lifecycle.

`Set` vs. `WeakSet`

Similar to `Map`, `Set`s allow any data type as keys. The elements within a `Set` must be unique. `Set`s are useful for storing unique values and checking for membership efficiently. Common use cases include removing duplicates from arrays or keeping track of completed tasks.

On the other hand, `WeakSet` only allows objects as elements, and these object elements are held weakly, similar to `WeakMap` keys. `WeakSet`s are less commonly used, but applicable when you want a collection of unique objects without affecting their garbage collection. This might be necessary for:

- Tracking DOM nodes that have been interacted with without affecting their memory management.
- Implementing custom object weak references for specific use cases.

Here's a table summarizing the key differences:

Feature	Map	WeakMap	Set	WeakSet
Key Types	Any data type	Objects (weak references)	Any data type (unique)	Objects (weak references, unique)
Garbage Collection	Keys and values are not garbage collected	Keys can be garbage collected if not referenced elsewhere	Elements are not garbage collected	Elements can be garbage collected if not referenced elsewhere
Use Cases	General-purpose	Caching, private DOM	Removing duplicates,	Object weak references,

Feature	Map	WeakMap	Set	WeakSet
	key-value storage	node data	membership checks	custom use cases

Choosing between them

- Use `Map` and `Set` for most scenarios where you need to store key-value pairs or unique elements and want to maintain references to both the keys/elements and the values.
- Use `WeakMap` and `WeakSet` cautiously in specific situations where you want to associate data with objects without affecting their garbage collection. Be aware of the implications of weak references and potential memory leaks if not used correctly.

How do you convert a `Set` to an array in JavaScript?

To convert a `Set` to an array in JavaScript, you can use the `Array.from()` method or the spread operator. For example:

```
const mySet = new Set([1, 2, 3]);
const myArray = Array.from(mySet);
// OR const myArray = [...mySet];

console.log(myArray); // Output: [1, 2, 3]
```

What is the difference between a `Map` object and a plain object in JavaScript?

Both `Map` objects and plain objects in JavaScript can store key-value pairs, but they have several key differences:

Feature	`Map`	Plain object
Key type	Any data type	String (or Symbol)
Key order	Maintained	Not guaranteed
Size property	Yes (`size`)	None

Feature	`Map`	Plain object
Iteration	`forEach`, `keys()`, `values()`, `entries()`	`for...in`, `Object.keys()`, etc.
Inheritance	No	Yes
Performance	Generally better for larger datasets and frequent additions/deletions	Faster for small datasets and simple operations
Serializable	No	Yes

How do `Set`s and `Map`s handle equality checks for objects?

`Set`s and `Map`s in JavaScript handle equality checks for objects based on reference equality, not deep equality. This means that two objects are considered equal only if they reference the same memory location. For example, if you add two different object literals with the same properties to a `Set`, they will be treated as distinct entries.

```
const set = new Set();
const obj1 = { a: 1 };
const obj2 = { a: 1 };

set.add(obj1);
set.add(obj2);

console.log(set.size); // Output: 2
```

What are some common performance bottlenecks in JavaScript applications?

Common performance bottlenecks in JavaScript applications include inefficient DOM manipulation, excessive use of global variables, blocking the main thread with heavy computations, memory leaks, and improper use of asynchronous operations. To mitigate these issues, you can use techniques like debouncing and throttling, optimizing DOM updates, and leveraging web workers for heavy computations.

Explain the concept of debouncing and throttling

Debouncing and throttling are techniques used to control the rate at which a function is executed. Debouncing ensures that a function is only called after a specified delay has passed since the last time it was invoked. Throttling ensures that a function is called at most once in a specified time interval.

Debouncing delays the execution of a function until a certain amount of time has passed since it was last called. This is useful for scenarios like search input fields where you want to wait until the user has stopped typing before making an API call.

```
function debounce(func, delay) {
  let timeoutId;
  return function (...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => func.apply(this, args), delay);
  };
}

const debouncedHello = debounce(
    () => console.log('Hello world!'),
2000);

debouncedHello(); // Prints 'Hello world!' after 2 seconds
```

Throttling ensures that a function is called at most once in a specified time interval. This is useful for scenarios like window resizing or scrolling where you want to limit the number of times a function is called.

```
function throttle(func, limit) {
  let inThrottle;
  return function (...args) {
    if (!inThrottle) {
      func.apply(this, args);
      inThrottle = true;
      setTimeout(() => (inThrottle = false), limit);
    }
  };
}

const handleResize = throttle(() => {
  // Update element positions
  console.log('Window resized at', new Date().toLocaleTimeString());
}, 2000);

// Simulate rapid calls to handleResize every 100ms
```

```
let intervalId = setInterval(() => {
  handleResize();
}, 100);
// 'Window resized' is outputted only every 2 seconds due to throttling
```

How can you optimize DOM manipulation for better performance?

To optimize DOM manipulation for better performance, minimize direct DOM access and updates. Use techniques like batching DOM changes, using `documentFragment` for multiple elements, and leveraging virtual DOM libraries like React. Also, consider using `requestAnimationFrame` for animations and avoid layout thrashing by reading and writing DOM properties separately.

What are some techniques for reducing reflows and repaints?

To reduce reflows and repaints, you can minimize DOM manipulations, batch DOM changes, use CSS classes for style changes, avoid complex CSS selectors, and use `requestAnimationFrame` for animations. Additionally, consider using `will-change` for elements that will change frequently and avoid layout thrashing by reading and writing to the DOM separately.

Explain the concept of lazy loading and how it can improve performance

Lazy loading is a design pattern that delays the loading of resources until they are actually needed. This can significantly improve performance by reducing initial load times and conserving bandwidth. For example, images on a webpage can be lazy-loaded so that they only load when they come into the viewport. This can be achieved using the `loading="lazy"` attribute in HTML or by using JavaScript libraries.

```
<img src="image.jpg" loading="lazy" alt="Lazy loaded image" />
```

What are Web Workers and how can they be used to improve performance?

Web Workers are a way to run JavaScript in the background, separate from the main execution thread of a web application. This helps in performing heavy computations without blocking the user interface. You can create a Web Worker using the `Worker` constructor and communicate with it using the `postMessage` and `onmessage` methods.

```
// main.js
const worker = new Worker('worker.js');
worker.postMessage('Hello, worker!');

worker.onmessage = function (event) {
  console.log('Message from worker:', event.data);
};

// worker.js
onmessage = function (event) {
  console.log('Message from main script:', event.data);
  postMessage('Hello, main script!');
};
```

Explain the concept of caching and how it can be used to improve performance

Caching is a technique used to store copies of files or data in a temporary storage location to reduce the time it takes to access them. It improves performance by reducing the need to fetch data from the original source repeatedly. In front end development, caching can be implemented using browser cache, service workers, and HTTP headers like `Cache-Control`.

What are some tools that can be used to measure and analyze JavaScript performance?

To measure and analyze JavaScript performance, you can use tools like Chrome DevTools, Lighthouse, WebPageTest, and JSPerf. Chrome DevTools provides a Performance panel for profiling, Lighthouse offers audits for performance metrics, WebPageTest allows for detailed performance testing, and JSPerf helps in comparing the performance of different JavaScript snippets.

How can you optimize network requests for better performance?

To optimize network requests for better performance, you can minimize the number of requests, use caching, compress data, and leverage modern web technologies like

HTTP/2 and service workers. For example, you can combine multiple CSS files into one to reduce the number of requests, use `Cache-Control` headers to cache static assets, and enable Gzip compression on your server to reduce the size of the data being transferred.

What are the different types of testing in software development?

In software development, there are several types of testing to ensure the quality and functionality of the application. These include unit testing, integration testing, system testing, and acceptance testing. Unit testing focuses on individual components, integration testing checks the interaction between components, system testing evaluates the entire system, and acceptance testing ensures the software meets user requirements.

Explain the difference between unit testing, integration testing, and end-to-end testing

Unit testing focuses on testing individual components or functions in isolation to ensure they work as expected. Integration testing checks how different modules or services work together. End-to-end testing simulates real user scenarios to verify the entire application flow from start to finish.

What are some popular JavaScript testing frameworks?

Some popular JavaScript testing frameworks include Jest, Mocha, Jasmine, and Cypress. Jest is known for its simplicity and integration with React. Mocha is highly flexible and often used with other libraries like Chai for assertions. Jasmine is a behavior-driven development framework that requires no additional libraries. Cypress is an end-to-end testing framework that provides a great developer experience.

How do you write unit tests for JavaScript code?

To write unit tests for JavaScript code, you typically use a testing framework like Jest or Mocha. First, you set up your testing environment by installing the necessary libraries. Then, you write test cases using functions like `describe`, `it`, or `test` to define your tests. Each test case should focus on a small, isolated piece of functionality. You use assertions to check if the output of your code matches the expected result.

Example using Jest:

```
// sum.js
function sum(a, b) {
  return a + b;
}
module.exports = sum;

// sum.test.js
const sum = require('./sum');

test('adds 1 + 2 to equal 3', () => {
  expect(sum(1, 2)).toBe(3);
});
```

Explain the concept of test-driven development (TDD)

Test-driven development (TDD) is a software development approach where you write tests before writing the actual code. The process involves writing a failing test, writing the minimum code to pass the test, and then refactoring the code while keeping the tests passing. This ensures that the code is always tested and helps in maintaining high code quality.

What are mocks and stubs and how are they used in testing?

Mocks and stubs are tools used in testing to simulate the behavior of real objects. Stubs provide predefined responses to function calls, while mocks are more complex and can verify interactions, such as whether a function was called and with what arguments. Stubs are used to isolate the code being tested from external dependencies, and mocks are used to ensure that the code interacts correctly with those dependencies.

How can you test asynchronous code in JavaScript?

To test asynchronous code in JavaScript, you can use testing frameworks like Jest or Mocha. These frameworks provide built-in support for handling asynchronous operations. You can use `async` / `await` or return promises in your test functions. For example, in Jest, you can write:

```
test('fetches data successfully', async () => {
  const data = await fetchData();
  expect(data).toBeDefined();
});
```

Alternatively, you can use callbacks and the `done` function to signal the end of an asynchronous test.

What are some best practices for writing maintainable and effective tests in JavaScript?

To write maintainable and effective tests, ensure they are clear, concise, and focused on a single behavior. Use descriptive names for test cases and avoid hardcoding values. Mock external dependencies and keep tests isolated. Regularly review and refactor tests to keep them up-to-date with the codebase.

Explain the concept of code coverage and how it can be used to assess test quality

Code coverage is a metric that measures the percentage of code that is executed when the test suite runs. It helps in assessing the quality of tests by identifying untested parts of the codebase. Higher code coverage generally indicates more thorough testing, but it doesn't guarantee the absence of bugs. Tools like Istanbul or Jest can be used to measure code coverage.

What are some tools that can be used for JavaScript testing?

For JavaScript testing, you can use tools like Jest, Mocha, Jasmine, and Cypress. Jest is popular for its ease of use and built-in features. Mocha is flexible and can be paired with other libraries. Jasmine is known for its simplicity and behavior-driven development (BDD) style. Cypress is great for end-to-end testing with a focus on real browser interactions.

What are design patterns and why are they useful?

Design patterns are reusable solutions to common problems in software design. They provide a template for how to solve a problem that can be used in many different situations. They are useful because they help developers avoid common pitfalls, improve code readability, and make it easier to maintain and scale applications.

Explain the concept of the Singleton pattern

The Singleton pattern ensures that a class has only one instance and provides a global point of access to that instance. This is useful when exactly one object is needed to coordinate actions across the system. In JavaScript, this can be implemented using closures or ES6 classes.

```
class Singleton {
  constructor() {
    if (!Singleton.instance) {
      Singleton.instance = this;
    }
    return Singleton.instance;
  }
}

const instance1 = new Singleton();
const instance2 = new Singleton();

console.log(instance1 === instance2); // true
```

What is the Factory pattern and how is it used?

The Factory pattern is a design pattern used to create objects without specifying the exact class of the object that will be created. It provides a way to encapsulate the instantiation logic and can be particularly useful when the creation process is complex or when the type of object to be created is determined at runtime.

For example, in JavaScript, you can use a factory function to create different types of objects:

```
function createAnimal(type) {
  if (type === 'dog') {
    return { sound: 'woof' };
  } else if (type === 'cat') {
    return { sound: 'meow' };
  }
}
```

```
const dog = createAnimal('dog');
const cat = createAnimal('cat');
```

Explain the Observer pattern and its use cases

The Observer pattern is a design pattern where an object, known as the subject, maintains a list of its dependents, called observers, and notifies them of any state changes. This pattern is useful for implementing distributed event-handling systems, such as updating the user interface in response to data changes or implementing event-driven architectures.

What is the Module pattern and how does it help with encapsulation?

The Module pattern in JavaScript is a design pattern used to create self-contained modules of code. It helps with encapsulation by allowing you to define private and public members within a module. Private members are not accessible from outside the module, while public members are exposed through a returned object. This pattern helps in organizing code, avoiding global namespace pollution, and maintaining a clean separation of concerns.

```
var myModule = (function () {
  var privateVar = 'I am private';

  function privateMethod() {
    console.log(privateVar);
  }

  return {
    publicMethod: function () {
      privateMethod();
    },
  };
})();

myModule.publicMethod(); // Logs: I am private
```

Explain the concept of the Prototype pattern

The Prototype pattern is a creational design pattern used to create new objects by copying an existing object, known as the prototype. This pattern is useful when the cost of creating a new object is more expensive than cloning an existing one. In JavaScript, this can be achieved using the `Object.create` method or by using the `prototype` property of a constructor function.

```
const prototypeObject = {
  greet() {
    console.log('Hello, world!');
  },
};

const newObject = Object.create(prototypeObject);
newObject.greet(); // Outputs: Hello, world!
```

What is the Decorator pattern and how is it used?

The Decorator pattern is a structural design pattern that allows behavior to be added to individual objects, dynamically, without affecting the behavior of other objects from the same class. It is used to extend the functionalities of objects by wrapping them with additional behavior. In JavaScript, this can be achieved using higher-order functions or classes.

For example, if you have a `Car` class and you want to add features like `GPS` or `Sunroof` without modifying the `Car` class itself, you can create decorators for these features.

```
class Car {
  drive() {
    return 'Driving';
  }
}

class CarDecorator {
  constructor(car) {
    this.car = car;
  }

  drive() {
    return this.car.drive();
  }
}
```

```
class GPSDecorator extends CarDecorator {
  drive() {
    return `${super.drive()} with GPS`;
  }
}

const myCar = new Car();
const myCarWithGPS = new GPSDecorator(myCar);
console.log(myCarWithGPS.drive()); // "Driving with GPS"
```

Explain the concept of the Strategy pattern

The Strategy pattern is a behavioral design pattern that allows you to define a family of algorithms, encapsulate each one as a separate class, and make them interchangeable. This pattern lets the algorithm vary independently from the clients that use it. For example, if you have different sorting algorithms, you can define each one as a strategy and switch between them without changing the client code.

```
class Context {
  constructor(strategy) {
    this.strategy = strategy;
  }

  executeStrategy(data) {
    return this.strategy.doAlgorithm(data);
  }
}

class ConcreteStrategyA {
  doAlgorithm(data) {
    // Implementation of algorithm A
    return 'Algorithm A was run on ' + data;
  }
}

class ConcreteStrategyB {
  doAlgorithm(data) {
    // Implementation of algorithm B
    return 'Algorithm B was run on ' + data;
  }
}

// Usage
```

```
const context = new Context(new ConcreteStrategyA());
context.executeStrategy('someData');
// Output: Algorithm A was run on someData
```

What is the Command pattern and how is it used?

The Command pattern is a behavioral design pattern that turns a request into a stand-alone object containing all information about the request. This transformation allows for parameterization of methods with different requests, queuing of requests, and logging of the requests. It also supports undoable operations. In JavaScript, it can be implemented by creating command objects with `execute` and `undo` methods.

```
class Command {
  execute() {}
  undo() {}
}

class LightOnCommand extends Command {
  constructor(light) {
    super();
    this.light = light;
  }
  execute() {
    this.light.on();
  }
  undo() {
    this.light.off();
  }
}

class Light {
  on() {
    console.log('Light is on');
  }
  off() {
    console.log('Light is off');
  }
}

const light = new Light();
const lightOnCommand = new LightOnCommand(light);
```

```
lightOnCommand.execute(); // Light is on
lightOnCommand.undo(); // Light is off
```

Why is extending built-in JavaScript objects not a good idea?

Extending a built-in/native JavaScript object means adding properties/functions to its `prototype`. While this may seem like a good idea at first, it is dangerous in practice. Imagine your code uses a few libraries that both extend the `Array.prototype` by adding the same `contains` method, the implementations will overwrite each other and your code will have unpredictable behavior if these two methods do not work the same way.

The only time you may want to extend a native object is when you want to create a polyfill, essentially providing your own implementation for a method that is part of the JavaScript specification but might not exist in the user's browser due to it being an older browser.

What is Cross-Site Scripting (XSS) and how can you prevent it?

Cross-Site Scripting (XSS) is a security vulnerability that allows attackers to inject malicious scripts into web pages viewed by other users. This can lead to data theft, session hijacking, and other malicious activities. To prevent XSS, you should validate and sanitize user inputs, use Content Security Policy (CSP), and escape data before rendering it in the browser.

Explain the concept of Cross-Site Request Forgery (CSRF) and its mitigation techniques

Cross-Site Request Forgery (CSRF) is an attack where a malicious website tricks a user's browser into making an unwanted request to another site where the user is authenticated. This can lead to unauthorized actions being performed on behalf of the user. Mitigation techniques include using anti-CSRF tokens, SameSite cookies, and ensuring proper CORS configurations.

How can you prevent SQL injection vulnerabilities in JavaScript applications?

To prevent SQL injection vulnerabilities in JavaScript applications, always use parameterized queries or prepared statements instead of string concatenation to construct SQL queries. This ensures that user input is treated as data and not executable code. Additionally, use ORM libraries that handle SQL injection prevention for you, and always validate and sanitize user inputs.

What are some best practices for handling sensitive data in JavaScript?

Handling sensitive data in JavaScript requires careful attention to security practices. Avoid storing sensitive data in client-side storage like localStorage or sessionStorage. Use HTTPS to encrypt data in transit. Implement proper authentication and authorization mechanisms. Sanitize and validate all inputs to prevent injection attacks. Consider using environment variables for sensitive data in server-side code.

Explain the concept of Content Security Policy (CSP) and how it enhances security

Content Security Policy (CSP) is a security feature that helps prevent various types of attacks, such as Cross-Site Scripting (XSS) and data injection attacks, by specifying which content sources are trusted. It works by allowing developers to define a whitelist of trusted sources for content like scripts, styles, and images. This is done through HTTP headers or meta tags. For example, you can use the `Content-Security-Policy` header to specify that only scripts from your own domain should be executed:

```
Content-Security-Policy: script-src 'self'
```

What are some common security headers and their purpose?

Security headers are HTTP response headers that help protect web applications from various attacks. Some common security headers include:

- `Content-Security-Policy (CSP)`: Prevents cross-site scripting (XSS) and other code injection attacks by specifying allowed content sources.
- `X-Content-Type-Options`: Prevents MIME type sniffing by instructing the browser to follow the declared `Content-Type`.
- `Strict-Transport-Security (HSTS)`: Enforces secure (HTTPS) connections to the server.

- `X-Frame-Options`: Prevents clickjacking by controlling whether a page can be displayed in a frame.
- `X-XSS-Protection`: Enables the cross-site scripting (XSS) filter built into most browsers.
- `Referrer-Policy`: Controls how much referrer information is included with requests.

How can you prevent clickjacking attacks?

To prevent clickjacking attacks, you can use the `X-Frame-Options` HTTP header to control whether your site can be embedded in iframes. Set it to `DENY` to prevent all framing, or `SAMEORIGIN` to allow framing only from the same origin. Additionally, you can use the `Content-Security-Policy` (CSP) header with the `frame-ancestors` directive to specify which origins are allowed to frame your content.

```
X-Frame-Options: DENY
```

```
Content-Security-Policy: frame-ancestors 'self'
```

Explain the concept of input validation and its importance in security

Input validation is the process of ensuring that user input is correct, safe, and meets the application's requirements. It is crucial for security because it helps prevent attacks like SQL injection, cross-site scripting (XSS), and other forms of data manipulation. By validating input, you ensure that only properly formatted data enters your system, reducing the risk of malicious data causing harm.

What are some tools and techniques for identifying security vulnerabilities in JavaScript code?

To identify security vulnerabilities in JavaScript code, you can use static code analysis tools like ESLint with security plugins, dynamic analysis tools like OWASP ZAP, and dependency checkers like npm audit. Additionally, manual code reviews and adhering to secure coding practices are essential techniques.

How can you implement secure authentication and authorization in JavaScript applications?

To implement secure authentication and authorization in JavaScript applications, use HTTPS to encrypt data in transit, and store sensitive data like tokens securely using `localStorage` or `sessionStorage`. Implement token-based authentication using JWTs, and validate tokens on the server side. Use libraries like OAuth for third-party authentication and ensure proper role-based access control (RBAC) for authorization.

Explain the same-origin policy with regards to JavaScript

The same-origin policy is a security measure implemented in web browsers to prevent malicious scripts on one page from accessing data on another page. It ensures that web pages can only make requests to the same origin, where the origin is defined by the combination of the protocol, domain, and port. For example, a script from `http://example.com` cannot access data from `http://anotherdomain.com`.

What is `'use strict';` in JavaScript for?

`'use strict'` is a statement used to enable strict mode to entire scripts or individual functions. Strict mode is a way to opt into a restricted variant of JavaScript.

Advantages

- Makes it impossible to accidentally create global variables.
- Makes assignments which would otherwise silently fail to throw an exception.
- Makes attempts to delete undeletable properties throw an exception (where before the attempt would simply have no effect).
- Requires that function parameter names be unique.
- `this` is `undefined` in the global context.
- It catches some common coding bloopers, throwing exceptions.
- It disables features that are confusing or poorly thought out.

Disadvantages

- Many missing features that some developers might be used to.
- No more access to `function.caller` and `function.arguments`.
- Concatenation of scripts written in different strict modes might cause issues.

Overall, the benefits outweigh the disadvantages and there is not really a need to rely on the features that strict mode prohibits. We should all be using strict mode by default.

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

Some of the most commonly used tools and techniques for debugging JavaScript:

- JavaScript language
    - `console` methods (e.g. `console.log()`, `console.error()`, `console.warn()`, `console.table()`)
    - `debugger` statement
- Breakpoints (browser or IDE)
- JavaScript frameworks
    - React Devtools
    - Redux Devtools
    - Vue Devtools
- Browser developer tools
    - **Chrome DevTools**: The most widely used tool for debugging JavaScript. It provides a rich set of features including the ability to set breakpoints, inspect variables, view the call stack, and more.
    - **Firefox Developer Tools**: Similar to Chrome DevTools with its own set of features for debugging.
    - **Safari Web Inspector**: Provides tools for debugging on Safari.
    - **Edge Developer Tools**: Similar to Chrome DevTools, as Edge is now Chromium-based.
- Network requests
    - **Postman**: Useful for debugging API calls.
    - **Fiddler**: Helps capture and inspect HTTP/HTTPS traffic.
    - **Charles Proxy**: Another tool for intercepting and debugging network calls.

How does JavaScript garbage collection work?

Garbage collection in JavaScript is an automatic memory management mechanism that reclaims memory occupied by objects and variables that are no longer in use by the program. The two most common algorithms are mark-and-sweep and generational garbage collection.

Mark-and-sweep

The most common garbage collection algorithm used in JavaScript is the Mark-and-sweep algorithm. It operates in two phases:

- **Marking phase**: The garbage collector traverses the object graph, starting from the root objects (global variables, currently executing functions, etc.), and marks all reachable objects as "in-use".
- **Sweeping phase**: The garbage collector sweeps through memory, removing all unmarked objects, as they are considered unreachable and no longer needed.

This algorithm effectively identifies and removes objects that have become unreachable, freeing up memory for new allocations.

Generational garbage collection

Leveraged by modern JavaScript engines, objects are divided into different generations based on their age and usage patterns. Frequently accessed objects are moved to younger generations, while less frequently used objects are promoted to older generations. This optimization reduces the overhead of garbage collection by focusing on the younger generations, where most objects are short-lived.

Different JavaScript engines (differs according to browsers) implement different garbage collection algorithms and there's no standard way of doing garbage collection.

Explain what a single page app is and how to make one SEO-friendly

A single page application (SPA) is a web application that loads a single HTML page and dynamically updates content as the user interacts with the app. This approach provides a more fluid user experience but can be challenging for SEO because search engines may not execute JavaScript to render content. To make an SPA SEO-friendly, you can use server-side rendering (SSR) or static site generation (SSG) to ensure that search engines can index your content. Tools like Next.js for React or Nuxt.js for Vue.js can help achieve this.

How can you share code between JavaScript files?

To share code between JavaScript files, you can use modules. In modern JavaScript, you can use ES6 modules with `export` and `import` statements. For example, you can export a function from one file and import it into another:

```
// file1.js
export function greet() {
  console.log('Hello, world!');
}
```

```
// file2.js
import { greet } from './file1.js';
greet();
```

Alternatively, in Node.js, you can use `module.exports` and `require`:

```
// file1.js
module.exports = function greet() {
  console.log('Hello, world!');
};

// file2.js
const greet = require('./file1.js');
greet();
```

How do you organize your code?

I organize my code by following a modular approach, using a clear folder structure, and adhering to coding standards and best practices. I separate concerns by dividing code into different layers such as components, services, and utilities. I also use naming conventions and documentation to ensure code readability and maintainability.

What are some of the advantages/disadvantages of writing JavaScript code in a language that compiles to JavaScript?

Using languages that compile to JavaScript, like TypeScript or CoffeeScript, can offer several advantages such as improved syntax, type safety, and better tooling. However, they also come with disadvantages like added build steps, potential performance overhead, and the need to learn new syntax.

Advantages:

- Improved syntax and readability
- Type safety and error checking
- Better tooling and editor support

Disadvantages:

- Added build steps and complexity

- Potential performance overhead
- Learning curve for new syntax

When would you use `document.write()`?

`document.write()` is rarely used in modern web development because it can overwrite the entire document if called after the page has loaded. It is mainly used for simple tasks like writing content during the initial page load, such as for educational purposes or quick debugging. However, it is generally recommended to use other methods like `innerHTML`, `appendChild()`, or modern frameworks for manipulating the DOM.