JavaScript Interview Questions and Answers

Contents

Fundamentals

1. What is JavaScript?

JavaScript is a high-level, interpreted programming language primarily used for creating interactive effects within web browsers. It's a core technology of the World Wide Web alongside HTML and CSS, allowing developers to create dynamic web content, control multimedia, animate images, and more. It now also runs on the server-side (Node.js) and in various other environments.

2. What are the key features of JavaScript?

- **Dynamic typing**: Variables can change types
- **Prototype-based object orientation**
- **First-class functions**: Functions can be assigned to variables, passed as arguments, and returned from other functions
- **Event-driven programming**
- **Single-threaded with asynchronous capabilities**
- **Closures**
- **JSON support**
- **ES6+ features**: Arrow functions, classes, modules, promises, etc.

3. What's the difference between JavaScript and Java?

Despite the similar names, JavaScript and Java are entirely different languages:

- **Origin**: JavaScript was created by Brendan Eich at Netscape; Java by Sun Microsystems (now Oracle)
- **Execution**: JavaScript is interpreted (JIT compiled in modern engines); Java is compiled to bytecode
- **Typing**: JavaScript is dynamically typed; Java is statically typed
- **Object model**: JavaScript uses prototype-based inheritance; Java uses class-based inheritance
- **Purpose**: JavaScript was initially created for web browsers; Java was designed as a general-purpose language

4. What is ECMAScript?

ECMAScript is the standardized specification that JavaScript implements. It defines the core features and syntax of the language, which browser vendors and other environments then implement. JavaScript is essentially the most popular implementation of the ECMAScript standard. ECMAScript versions (ES6, ES2022, etc.) define new features and improvements to the language.

5. How do you declare variables in JavaScript?

Variables in JavaScript can be declared using:

```
var name; // Function-scoped, hoisted
let name; // Block-scoped
const name; // Block-scoped, cannot be reassigned
```

6. What is the difference between var, let, and const?

- **var**: Function-scoped, hoisted to the top of its scope, can be redeclared, can be updated
- **let**: Block-scoped, not hoisted, cannot be redeclared in the same scope, can be updated
- **const**: Block-scoped, not hoisted, cannot be redeclared in the same scope, cannot be reassigned after initialization (though object properties can still be modified)

7. What are primitive data types in JavaScript?

JavaScript has seven primitive data types:

- **String**: Textual data
- **Number**: Integer or floating-point numbers
- **Boolean**: true or false
- **null**: Intentional absence of any object value
- **undefined**: A variable that has been declared but not assigned a value
- **Symbol**: Unique and immutable value (ES6)
- **BigInt**: Integers of arbitrary precision (ES2020)

8. What is hoisting in JavaScript?

Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope during the compilation phase. This means that variables and function declarations can be used before they are actually written in the code. However, only the declarations are hoisted, not initializations. Variables declared with `var` are hoisted and initialized with `undefined`, while `let` and `const` are hoisted but not initialized (they're in the "temporal dead zone" until their declaration).

9. Explain the concept of scope in JavaScript.

Scope determines the accessibility of variables, functions, and objects in particular parts of your code:

- **Global scope**: Variables declared outside any function or block are globally accessible
- **Function scope**: Variables declared within a function are accessible only within that function
- **Block scope**: Variables declared within a block (e.g., within if statements or loops) using `let` or `const` are accessible only within that block
- **Lexical scope**: Inner functions have access to variables declared in their outer (parent) functions

10. What is temporal dead zone?

The Temporal Dead Zone (TDZ) is the period between entering a scope where a variable is declared using `let` or `const` and the actual declaration. During this period, accessing the variable will result in a `ReferenceError`. This behavior enforces better coding practices by preventing the use of variables before they are declared.

11. What are closures in JavaScript?

A closure is a function that has access to its own scope, the outer function's variables, and global variables, even after the outer function has returned. Closures remember the environment in which they were created. They're used for data privacy, factory functions, and maintaining state in asynchronous code.

```
function createCounter() {
  let count = 0;
  return function() {
    return ++count;
  };
}

const counter = createCounter();
console.log(counter()); // 1
console.log(counter()); // 2
```

12. How does the this keyword work in JavaScript?

The `this` keyword refers to the object that is executing the current function:

- In a method, `this` refers to the owner object
- In a regular function, `this` refers to the global object (window in browsers, global in Node.js) or undefined in strict mode
- In an event handler, `this` refers to the element that received the event
- In arrow functions, `this` retains the value of the enclosing lexical context
- With `call()`, `apply()`, or `bind()`, `this` can be explicitly set

13. What is event bubbling and event capturing?

These are two phases of event propagation in the DOM:

- **Event capturing (trickling)**: The event starts from the root and moves down to the target element
- **Event bubbling**: After reaching the target, the event bubbles up from the target to the root

When an event occurs on an element, handlers first run during the capturing phase, then during the bubbling phase. Most event handlers are registered for the bubbling phase by default.

14. Explain the event delegation pattern.

Event delegation is a technique where you attach a single event listener to a parent element instead of multiple listeners on individual child elements. It takes advantage of event bubbling, where events triggered on child elements bubble up to their parents. This approach is more efficient for dynamic content and reduces memory usage.

```
// Instead of adding click handlers to each button
document.getElementById('parent-container').addEventListener('click',
function(e) {
  if (e.target.className === 'btn') {
    // Handle the button click
  }
});
```

15. What is the difference between == and ===?

- **==** (Loose equality): Compares values after type conversion (coercion)
- **===** (Strict equality): Compares both values and types without type conversion

```
"5" == 5  // true (string "5" is converted to number 5)
"5" === 5 // false (different types: string vs number)
```

16. What is type coercion in JavaScript?

Type coercion is the automatic or implicit conversion of values from one data type to another. JavaScript performs type coercion when different types are used in operations:

```
"5" + 2     // "52" (number is coerced to string for concatenation)
"5" - 2     // 3 (string is coerced to number for subtraction)
true + 1    // 2 (boolean is coerced to number: true becomes 1)
if ("hello") // non-empty string is coerced to true in boolean context
```

17. How does JavaScript handle asynchronous code?

JavaScript handles asynchronous operations using:

- **Callbacks**: Functions passed as arguments to be executed later
- **Promises**: Objects representing the eventual completion or failure of an asynchronous operation
- **Async/await**: Syntactic sugar on top of promises for cleaner asynchronous code
- **Event loop**: Orchestrates the execution of code, handling the callback queue and microtasks

These mechanisms allow JavaScript to execute non-blocking operations despite being single-threaded.

18. What are callback functions?

A callback function is a function passed as an argument to another function, which is then invoked inside the outer function to complete an action. Callbacks are commonly used for asynchronous operations like API requests, timers, and event handlers.

```
function fetchData(callback) {
  // Simulating an API call
  setTimeout(() => {
    const data = { name: "John", age: 30 };
    callback(data);
  }, 1000);
}
```

```
fetchData(function(data) {
  console.log(data); // Executed after the data is "fetched"
});
```

19. Explain the event loop in JavaScript.

The event loop is the mechanism that allows JavaScript to perform non-blocking operations despite being single-threaded. It works by:

1. Executing the code in the call stack
2. Moving async operations to Web APIs (in browsers) or C++ APIs (in Node.js)
3. Placing callbacks in the callback queue when they're ready to execute
4. When the call stack is empty, moving the first callback from the queue to the stack (this is called the "tick" of the event loop)
5. Prioritizing microtasks (like Promise callbacks) over regular tasks

This process enables JavaScript to handle many operations concurrently without blocking the main thread.

20. What is the difference between synchronous and asynchronous programming?

- **Synchronous programming**: Code executes sequentially, with each operation completing before the next one begins. This can block the execution flow if operations take time.
- **Asynchronous programming**: Operations can be executed without blocking the main thread. The program continues to run while waiting for operations to complete, using callbacks, promises, or async/await to handle the results when ready.

Functions

21. What are the different ways to create functions in JavaScript?

- **Function declarations**: `function name() {}`
- **Function expressions**: `const name = function() {}`
- **Arrow functions**: `const name = () => {}`
- **Method definitions**: `const obj = { name() {} }`
- **Constructor functions**: `const fn = new Function('a', 'b', 'return a + b')`
- **IIFE (Immediately Invoked Function Expression)**: `(function() {})()`
- **Generator functions**: `function* name() {}`

22. What is a function declaration vs. function expression?

- **Function declaration**: Defines a named function and is hoisted entirely
- `function add(a, b) {`
- `  return a + b;`

- `}`
- **Function expression**: Defines a function as part of an expression and is not hoisted (only the variable declaration is hoisted if using `var`)
- `const add = function(a, b) {`
- `  return a + b;`
- `};`

23. What are arrow functions and how do they differ from regular functions?

Arrow functions are a concise syntax introduced in ES6:

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

Key differences from regular functions:

- No `this` binding (inherit `this` from their surrounding scope)
- No `arguments` object
- Cannot be used as constructors (no `new` keyword)
- No `super` or `new.target`
- Cannot be used as generator functions
- More concise syntax for simple functions

24. What is function hoisting?

Function hoisting allows function declarations to be used before they're defined in the code. During the compilation phase, function declarations are moved to the top of their scope:

```
// This works because of hoisting
sayHello(); // "Hello"

function sayHello() {
  console.log("Hello");
}

// Function expressions are not hoisted the same way
tryThis(); // Error: tryThis is not a function

var tryThis = function() {
  console.log("Try this");
};
```

25. What are higher-order functions?

Higher-order functions are functions that either:

1. Take one or more functions as arguments, or
2. Return a function as their result

Examples include `map()`, `filter()`, `reduce()`, and function factories. They enable functional programming patterns in JavaScript.

```
// map is a higher-order function that takes a function as an argument
const doubled = [1, 2, 3].map(num => num * 2);

// Function that returns another function
function multiplier(factor) {
  return function(number) {
    return number * factor;
  };
}

const double = multiplier(2);
double(5); // 10
```

26. What is a pure function?

A pure function:

1. Always returns the same output for the same inputs
2. Has no side effects (doesn't modify external state or variables)
3. Doesn't rely on external state

Pure functions are easier to test, debug, and reason about. They're a fundamental concept in functional programming.

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

// Impure function (uses external state)
let total = 0;
function addToTotal(value) {
  total += value;
  return total;
}
```

27. What is a callback function?

A callback function is a function passed as an argument to another function, which is then invoked inside the outer function. Callbacks are commonly used for asynchronous operations, event handlers, and higher-order functions.

```
function doSomethingAsync(callback) {
  setTimeout(() => {
    const result = "operation complete";
    callback(result);
  }, 1000);
}

doSomethingAsync(result => console.log(result));
```

28. What is function currying?

Currying is the technique of converting a function that takes multiple arguments into a sequence of functions that each take a single argument. This allows for partial application of functions and creating more specialized functions from general ones.

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

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

// Usage
add(1, 2, 3); // 6
curriedAdd(1)(2)(3); // 6

// Partial application
const add1 = curriedAdd(1);
const add1And2 = add1(2);
add1And2(3); // 6
```

29. What is a generator function?

Generator functions allow you to define an iterative algorithm by writing a function whose execution can be paused and resumed. They use the `function*` syntax and the `yield` keyword to define points where the function can yield values and later resume.

```
function* countUp() {
  let count = 0;
  while (true) {
    yield count++;
  }
}

const counter = countUp();
console.log(counter.next().value); // 0
console.log(counter.next().value); // 1
console.log(counter.next().value); // 2
```

30. How do you handle default parameters in JavaScript functions?

ES6 introduced default parameters, allowing function parameters to have default values if no value or `undefined` is passed:

```
function greet(name = "Guest", greeting = "Hello") {
  return `${greeting}, ${name}!`;
}
```

```
greet(); // "Hello, Guest!"
greet("John"); // "Hello, John!"
greet("Jane", "Welcome"); // "Welcome, Jane!"
```

31. What are rest parameters in functions?

Rest parameters, introduced in ES6, allow you to represent an indefinite number of arguments as an array. They're indicated by three dots (`...`) followed by the parameter name:

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

sum(1, 2, 3, 4); // 10
```

32. What is method chaining?

Method chaining is a programming pattern where multiple methods are called in sequence on the same object, with each method returning the object itself (`this`). This allows for more concise and readable code.

```
class Calculator {
  constructor() {
    this.value = 0;
  }

  add(n) {
    this.value += n;
    return this;
  }

  subtract(n) {
    this.value -= n;
    return this;
  }

  multiply(n) {
    this.value *= n;
    return this;
  }

  getValue() {
    return this.value;
  }
}

// Method chaining
new Calculator()
  .add(5)
  .multiply(2)
  .subtract(3)
  .getValue(); // 7
```

33. What is recursion in JavaScript?

Recursion is when a function calls itself to solve a problem. It typically involves a base case to stop the recursion and a recursive case that makes progress toward the base case.

```
function factorial(n) {
  // Base case
  if (n <= 1) return 1;

  // Recursive case
  return n * factorial(n - 1);
}

factorial(5); // 120 (5 * 4 * 3 * 2 * 1)
```

34. What is memoization?

Memoization is an optimization technique that stores the results of expensive function calls and returns the cached result when the same inputs occur again. It's particularly useful for recursive functions and complex calculations.

```
function memoizedFibonacci() {
  const cache = {};

  return function fib(n) {
    if (n in cache) {
      return cache[n];
    }

    if (n <= 1) return n;

    const result = fib(n - 1) + fib(n - 2);
    cache[n] = result;
    return result;
  };
}

const fibonacci = memoizedFibonacci();
fibonacci(40); // Much faster than non-memoized version
```

35. What's the difference between function scoped and block scoped variables?

- **Function-scoped variables** (`var`): Accessible throughout the entire function in which they are declared, regardless of block boundaries
- **Block-scoped variables** (`let` and `const`): Accessible only within the block (denoted by `{}`) in which they are declared

```
function example() {
  var functionScoped = "I'm available throughout the function";

  if (true) {
    var alsoFunctionScoped = "I'm also available throughout the function";
    let blockScoped = "I'm only available in this if block";
    const alsoBlockScoped = "Me too";

    console.log(blockScoped); // Works
  }

  console.log(functionScoped); // Works
  console.log(alsoFunctionScoped); // Works
  console.log(blockScoped); // ReferenceError: blockScoped is not defined
}
```

36. What is an IIFE (Immediately Invoked Function Expression)?

An IIFE is a function that is defined and executed immediately after creation. It's often used to create a private scope for variables, avoiding polluting the global namespace.

```
// IIFE syntax
(function() {
  // Private scope
  var private = "This variable is not accessible outside";
  console.log("IIFE executed!");
})();

// With parameters
(function(name) {
  console.log(`Hello, ${name}!`);
})("John");
```

37. What is the arguments object in JavaScript functions?

The `arguments` object is an array-like object available within non-arrow functions that contains the values of all arguments passed to the function. It allows functions to accept an indefinite number of arguments.

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

sum(1, 2, 3, 4); // 10
```

Note: In modern JavaScript, rest parameters are generally preferred over the `arguments` object.

38. How do you handle variable scope inside loops with closures?

A common issue with closures in loops is that the closure captures the variable by reference, not by value. By the time the closure is executed, the loop variable may have changed:

```
// Problem
for (var i = 0; i < 3; i++) {
  setTimeout(() => console.log(i), 100); // Logs 3, 3, 3
}

// Solution 1: Use let instead of var for block scoping
for (let j = 0; j < 3; j++) {
  setTimeout(() => console.log(j), 100); // Logs 0, 1, 2
}

// Solution 2: Use an IIFE to create a new scope
for (var k = 0; k < 3; k++) {
  (function(index) {
    setTimeout(() => console.log(index), 100); // Logs 0, 1, 2
  })(k);
}
```

39. What is function composition?

Function composition is the process of combining two or more functions to produce a new function. It's a fundamental concept in functional programming, allowing you to build complex operations from simple ones.

```
// Basic function composition
const compose = (f, g) => x => f(g(x));

const addOne = x => x + 1;
const double = x => x * 2;

const addOneThenDouble = compose(double, addOne);
addOneThenDouble(3); // 8 (double(addOne(3)) = double(4) = 8)

// Multiple functions composition
const pipe = (...fns) => x => fns.reduce((y, f) => f(y), x);

const addOneThenDoubleThenSquare = pipe(addOne, double, x => x * x);
addOneThenDoubleThenSquare(3); // 64 ((3+1)*2)^2 = 8^2 = 64
```

40. What are the differences between function declarations and class declarations?

- **Function declarations**:
  - Hoisted completely (can be used before declaration)
  - Create a single function object
  - Use prototype for inheritance
- **Class declarations**:
  - Not hoisted (must be declared before use)
  - Provide a cleaner syntax for object-oriented programming
  - Always execute in strict mode
  - Methods are non-enumerable
  - Must use `new` when instantiating
  - Cannot be called as a function

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

```
Person.prototype.greet = function() {
  return `Hello, I'm ${this.name}`;
};

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

  greet() {
    return `Hello, I'm ${this.name}`;
  }
}
```

Objects and Prototypes

41. What are objects in JavaScript?

Objects in JavaScript are collections of key-value pairs, where values can be properties (data) or methods (functions). They are one of JavaScript's fundamental data types and are used to store various keyed collections and more complex entities.

```
// Object literal
const person = {
  name: "John",
  age: 30,
  greet: function() {
    return `Hello, I'm ${this.name}`;
  }
};
```

42. How do you create objects in JavaScript?

There are several ways to create objects in JavaScript:

1. **Object literals**:
2. `const obj = { key: "value" };`
3. **Constructor functions**:
4. `function Person(name) {`
5. `  this.name = name;`
6. `}`
7. `const person = new Person("John");`
8. **Object.create()**:
9. `const prototype = { greet() { return "Hello"; } };`
10. `const obj = Object.create(prototype);`
11. **ES6 Classes**:
12. `class Person {`
13. `  constructor(name) {`
14. `    this.name = name;`
15. `  }`
16. `}`
17. `const person = new Person("John");`
18. **Factory functions**:

```
function createPerson(name) {
  return { name };
}
const person = createPerson("John");
```

43. What is the prototype chain?

The prototype chain is JavaScript's mechanism for inheritance. When accessing a property or method on an object, JavaScript:

1. Looks for the property on the object itself
2. If not found, it looks on the object's prototype
3. If still not found, it looks on the prototype's prototype
4. This continues until either the property is found or the end of the chain is reached (null)

```
// Simple prototype chain
const animal = {
  makeSound: function() {
    return "Some sound";
  }
};

const dog = Object.create(animal);
dog.makeSound(); // "Some sound" (inherited from animal)

// Check prototype chain
Object.getPrototypeOf(dog) === animal; // true
```

44. How does inheritance work in JavaScript?

JavaScript uses prototypal inheritance rather than classical inheritance. Objects inherit directly from other objects through the prototype chain. There are several ways to implement inheritance:

1. **Constructor functions with prototypes**:

```
function Animal(name) {
  this.name = name;
}
Animal.prototype.makeSound = function() {
  return "Some sound";
};

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

// Set up inheritance
Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

// Add or override methods
Dog.prototype.makeSound = function() {
  return "Woof!";
};
```

22. **ES6 Classes** (syntactic sugar over prototypal inheritance):

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

  makeSound() {
    return "Some sound";
  }
}

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

  makeSound() {
    return "Woof!";
  }
}
```

43. **Object.create()**:

```
const animal = {
  init(name) {
    this.name = name;
    return this;
  },
  makeSound() {
    return "Some sound";
  }
};

const dog = Object.create(animal);
dog.makeSound = function() {
  return "Woof!";
};
```

45. What is prototypal inheritance?

Prototypal inheritance is a form of inheritance where objects inherit directly from other objects. In JavaScript, each object has an internal link to another object called its prototype. When trying to access a property or method that doesn't exist on an object, JavaScript looks for it on the object's prototype, then on the prototype's prototype, and so on up the chain.

This differs from classical inheritance (used in languages like Java or C++), where classes inherit from other classes, and objects are instances of classes.

```
// Simple example of prototypal inheritance
const parent = {
  greet() {
    return "Hello from parent";
  }
};

// child inherits from parent
const child = Object.create(parent);
child.name = "Child Object";

// child can use parent's methods
```

```
child.greet(); // "Hello from parent"
```

46. What is the difference between Object.create() and the constructor pattern?

- **Object.create()**:
  - Creates a new object with the specified prototype object
  - Provides direct inheritance without constructors
  - No initialization logic (unless manually added)
  - Clearer and more direct way to create objects with specific prototypes
- **Constructor pattern**:
  - Creates a function that is used with the `new` keyword
  - Automatically sets up prototype inheritance when used with `new`
  - Provides initialization logic in the constructor function
  - More traditional object-oriented style

```
// Object.create()
const personProto = {
  greet() {
    return `Hello, I'm ${this.name}`;
  }
};

const john = Object.create(personProto);
john.name = "John";

// Constructor pattern
function Person(name) {
  this.name = name;
}

Person.prototype.greet = function() {
  return `Hello, I'm ${this.name}`;
};

const jane = new Person("Jane");
```

47. What are factory functions?

Factory functions are functions that create and return objects without using the `new` keyword. They provide a way to create objects with private state through closures and avoid some of the complexities of constructors and prototypes.

```
function createPerson(name, age) {
  // Private variables
  const privateData = { birthYear: new Date().getFullYear() - age };

  // Public interface
  return {
    name,
    getAge() {
      return age;
    },
    getBirthYear() {
      return privateData.birthYear;
    },
    celebrate() {
      age++;
      privateData.birthYear--;
    }
  };
}

const john = createPerson("John", 30);
john.getAge(); // 30
john.celebrate();
john.getAge(); // 31
```

Advantages of factory functions:

- No need for `new` keyword
- Can create private variables and methods using closures
- No issues with `this` binding
- No prototype chain complexities

48. What are constructor functions?

Constructor functions are regular functions that are intended to be used with the `new` keyword to create and initialize objects. By convention, they start with a capital letter.

```
function Person(name, age) {
  // 'this' refers to the new object being created
  this.name = name;
  this.age = age;

  // Adding a method directly to the instance
  this.greet = function() {
    return `Hello, I'm ${this.name}`;
  };
}

// Adding a method to the prototype (more efficient)
Person.prototype.getAge = function() {
  return this.age;
};

// Creating instances
const john = new Person("John", 30);
john.greet(); // "Hello, I'm John"
```

When using `new` with a constructor function:

1. A new empty object is created
2. `this` is bound to the new object
3. The function body executes
4. The new object is returned (unless the function explicitly returns another object)

49. What is the new keyword and how does it work?

The `new` keyword is used to create instances of objects from constructor functions. When `new` is used, it performs the following steps:

1. Creates a new empty object
2. Sets the prototype of the new object to the constructor function's prototype property
3. Executes the constructor function with `this` bound to the new object
4. Returns the new object (unless the constructor explicitly returns a different object)

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

// This is what happens when you call 'new Person("John")'
function createInstance(constructor, ...args) {
  // 1. Create a new object with the constructor's prototype
  const instance = Object.create(constructor.prototype);

  // 2. Call the constructor with 'this' set to the new object
  const result = constructor.apply(instance, args);

  // 3. Return the new object, unless the constructor returned something
else
  return (typeof result === 'object' && result !== null) ? result :
instance;
}

const john = createInstance(Person, "John");
// Equivalent to: const john = new Person("John");
```

50. How do you add properties to an object?

Properties can be added to objects in several ways:

1. **During creation (object literal)**:
2. `const person = {`
3. `  name: "John",`
4. `  age: 30`
5. `};`
6. **Dot notation**:
7. `person.job = "Developer";`
8. **Bracket notation** (useful for dynamic property names):
9. `const propertyName = "skills";`
10. `person[propertyName] = ["JavaScript", "React"];`
11. **Object.defineProperty()** (for more control):
12. `Object.defineProperty(person, "salary", {`
13. `  value: 50000,`
14. `  writable: true,`
15. `  enumerable: true,`
16. `  configurable: true`
17. `});`
18. **Object.assign()** (to add multiple properties):
19. `Object.assign(person, {`
20. `  location: "New York",`
21. `  department: "Engineering"`
22. `});`

51. What are getters and setters?

Getters and setters are special methods that allow you to define how properties are accessed and modified. They let you execute code when getting or setting a property value, enabling validation, computed properties, and more.

```
const person = {
  firstName: "John",
  lastName: "Doe",

  // Getter
  get fullName() {
    return `${this.firstName} ${this.lastName}`;
  },

  // Setter
  set fullName(value) {
    const parts = value.split(' ');
    this.firstName = parts[0];
    this.lastName = parts[1];
  }
};

console.log(person.fullName); // "John Doe"
person.fullName = "Jane Smith";
console.log(person.firstName); // "Jane"
```

You can also use Object.defineProperty():

```
Object.defineProperty(person, 'age', {
  get() {
    return this._age;
  },
  set(value) {
    if (value < 0) {
      throw new Error('Age cannot be negative');
    }
    this._age = value;
  }
});
```

52. What is the difference between dot notation and bracket notation when accessing object properties?

- **Dot notation** (`object.property`):
  - More concise and readable
  - Property name must be a valid identifier (can't start with a number, no spaces or special characters)
  - Property name is literal (can't be a variable or expression)
- **Bracket notation** (`object['property']`):
  - Can use property names that aren't valid identifiers ("spaces allowed", "123", etc.)
  - Can use variables or expressions to compute the property name
  - Allows for dynamic property access

```
const person = {
  name: "John",
  "job title": "Developer",
  123: "numeric key"
};

// Dot notation
console.log(person.name); // "John"

// Bracket notation
console.log(person["name"]); // "John"
console.log(person["job title"]); // "Developer" (can't use dot notation here)
console.log(person[123]); // "numeric key"

// Dynamic property access
const key = "name";
console.log(person[key]); // "John"
```

53. How do you check if a property exists in an object?

There are several ways to check if a property exists in an object:

1. **Using the `in` operator**:
2. `const person = { name: "John", age: 30 };`
3. `console.log("name" in person); // true`
4. `console.log("job" in person); // false`
5. **Using `hasOwnProperty()`** (only checks own properties, not inherited ones):
6. `console.log(person.hasOwnProperty("name")); // true`
7. `console.log(person.hasOwnProperty("toString")); // false (inherited)`
8. **Direct comparison with `undefined`**:
9. `console.log(person.name !== undefined); // true`
10. `console.log(person.job !== undefined); // false`
11. **Using `Object.hasOwn()`** (modern alternative to hasOwnProperty):
12. `console.log(Object.hasOwn(person, "name")); // true`

54. What is property shadowing?

Property shadowing occurs when an object has a property with the same name as a property in its prototype chain. The object's own property "shadows" or overrides the property in the prototype when accessed.

```
const parent = {
  value: 42,
  getValue() {
    return this.value;
  }
};

const child = Object.create(parent);
console.log(child.getValue()); // 42 (inherited from parent)

// Shadow the parent's property
child.value = 100;
console.log(child.getValue()); // 100 (child's own property shadows parent's)
```

55. What is method borrowing?

Method borrowing (also called function borrowing) is a technique where you use methods from one object on another object. This is done using `call()`, `apply()`, or `bind()` to explicitly set the `this` value.

```
const person = {
  name: "John",
  greet() {
    return `Hello, I'm ${this.name}`;
  }
};

const anotherPerson = {
  name: "Jane"
};

// Borrow the greet method from person
person.greet.call(anotherPerson); // "Hello, I'm Jane"

// With apply (same as call but takes arguments as an array)
person.greet.apply(anotherPerson, []); // "Hello, I'm Jane"

// With bind (creates a new function bound to the specified this)
const janeGreet = person.greet.bind(anotherPerson);
janeGreet(); // "Hello, I'm Jane"
```

56. How do you clone an object in JavaScript?

There are several ways to clone an object in JavaScript:

1. **Shallow Clone**:
   - Using Object.assign():
   - `const original = { a: 1, b: { c: 2 } };`
   - `const clone = Object.assign({}, original);`
   - Using spread operator:
   - `const clone = { ...original };`
2. **Deep Clone**:
   - Using JSON (with limitations - doesn't handle functions, circular references, etc.):
   - `const deepClone = JSON.parse(JSON.stringify(original));`
   - Using a library like Lodash:
   - `const deepClone = _.cloneDeep(original);`
   - Using the structured clone algorithm:
   - `const deepClone = structuredClone(original);`

57. What is the difference between shallow and deep copy?

- **Shallow copy**: Creates a new object that has copies of the values of the original object. If the value is a reference to another object (nested object), only the reference is copied, not the referenced object itself. Changes to nested objects in the copy will affect the original.

- **Deep copy**: Creates a new object with copies of all values, including nested objects. Changes to nested objects in the deep copy will not affect the original.

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

// Shallow copy
const shallowCopy = { ...original };
shallowCopy.a = 10; // This doesn't affect original
shallowCopy.b.c = 20; // This DOES affect original
console.log(original.b.c); // 20

// Deep copy
const deepCopy = JSON.parse(JSON.stringify(original));
deepCopy.b.c = 30; // This doesn't affect original
console.log(original.b.c); // 20
```

58. What is Object destructuring?

Object destructuring is a JavaScript expression that allows you to extract properties from objects and bind them to variables with the same name or custom names.

```
// Basic destructuring
const person = { name: "John", age: 30, job: "Developer" };
const { name, age } = person;
console.log(name); // "John"
console.log(age); // 30

// With different variable names
const { name: personName, job: occupation } = person;
console.log(personName); // "John"
console.log(occupation); // "Developer"

// Default values
const { salary = 50000 } = person;
console.log(salary); // 50000 (default value since it doesn't exist in
person)

// Nested destructuring
const user = {
  id: 1,
  details: {
    firstName: "John",
    lastName: "Doe"
  }
};
const { details: { firstName, lastName } } = user;
console.log(firstName); // "John"

// Rest pattern
const { name, ...rest } = person;
console.log(rest); // { age: 30, job: "Developer" }
```

59. What is the difference between Object.freeze() and Object.seal()?

- **Object.freeze()**:

    - Makes an object immutable: properties cannot be added, removed, or changed
    - Property values cannot be modified (shallow freeze only - nested objects can still be modified)
    - Property attributes cannot be changed
- **Object.seal()**:
    - Prevents adding or removing properties
    - Existing properties can still be modified
    - Property attributes cannot be changed

```
// Object.freeze()
const frozen = Object.freeze({ name: "John", age: 30 });
frozen.age = 31; // Fails silently in non-strict mode, throws error in
strict mode
frozen.job = "Developer"; // Fails
delete frozen.name; // Fails
console.log(frozen); // { name: "John", age: 30 }

// Object.seal()
const sealed = Object.seal({ name: "Jane", age: 25 });
sealed.age = 26; // Works
sealed.job = "Developer"; // Fails
delete sealed.name; // Fails
console.log(sealed); // { name: "Jane", age: 26 }
```

60. How do you loop through objects in JavaScript?

There are several ways to loop through object properties:

1. **for...in loop** (includes inherited enumerable properties):

```
const person = { name: "John", age: 30, job: "Developer" };

for (const key in person) {
  if (Object.hasOwn(person, key)) { // Skip inherited properties
    console.log(`${key}: ${person[key]}`);
  }
}
```

9. **Object.keys()** (gets own enumerable property names):

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

13. **Object.values()** (gets own enumerable property values):

```
Object.values(person).forEach(value => {
  console.log(value);
});
```

17. **Object.entries()** (gets own enumerable [key, value] pairs):

```
Object.entries(person).forEach(([key, value]) => {
  console.log(`${key}: ${value}`);
});
```

Arrays and Array Methods

61. What are arrays in JavaScript?

Arrays in JavaScript are ordered collections of values. They are special objects with numeric indices and a length property. They can store any type of value, including other arrays or objects.

```
// Array creation
const numbers = [1, 2, 3, 4, 5];
const mixed = [1, "two", { three: 3 }, [4]];
const emptyArray = [];
const usingConstructor = new Array(3); // Creates array with length 3

// Accessing elements
console.log(numbers[0]); // 1
console.log(numbers[numbers.length - 1]); // 5
```

62. What array methods mutate the original array?

Several array methods modify the original array instead of returning a new one:

- **Adding/removing elements**:
  - `push()` - adds elements to the end
  - `pop()` - removes the last element
  - `unshift()` - adds elements to the beginning
  - `shift()` - removes the first element
  - `splice()` - adds/removes elements anywhere in the array
- **Rearranging elements**:
  - `sort()` - sorts the elements
  - `reverse()` - reverses the order of elements
- **Other mutating methods**:
  - `fill()` - fills the array with a static value
  - `copyWithin()` - copies part of an array to another location in the same array

Non-mutating methods include `concat()`, `slice()`, `map()`, `filter()`, `reduce()`, etc.

63. What are the ways to iterate over an array?

There are many ways to iterate over arrays in JavaScript:

1. **for loop**:
```
2. const arr = [1, 2, 3];
3. for (let i = 0; i < arr.length; i++) {
4.   console.log(arr[i]);
5. }
```
6. **forEach()**:
```
7. arr.forEach(item => {
8.   console.log(item);
9. });
```
10. **for...of loop**:
```
11.  for (const item of arr) {
12.    console.log(item);
13.  }
```
14. **map()** (returns a new array):
```
15.  arr.map(item => {
16.    console.log(item);
17.    return item * 2; // Transforms each item
18.  });
```

19. **for...in loop** (not recommended for arrays as it iterates over indices, including non-numeric properties):

```
20.  for (const index in arr) {
21.    console.log(arr[index]);
22.  }
```

23. **Using other array methods**:

```
24.  arr.reduce((acc, item) => {
25.    console.log(item);
26.    return acc;
27.  }, 0);
```

64. What is the difference between forEach() and map()?

- **forEach()**:
  - Executes a callback function for each element in the array
  - Always returns `undefined`
  - Cannot be chained with other array methods
  - Cannot be terminated early (except by throwing an error)
  - Used for side effects (like logging, modifying other variables)
- **map()**:
  - Creates a new array with the results of calling a callback on every element
  - Doesn't modify the original array
  - Returns a new array of the same length
  - Can be chained with other array methods
  - Used for transforming data

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

// forEach - for side effects
numbers.forEach(num => {
  console.log(num * 2);
}); // undefined

// map - for transformations
const doubled = numbers.map(num => num * 2);
console.log(doubled); // [2, 4, 6]
```

65. How do filter(), find(), and findIndex() work?

These methods are used for searching elements in arrays:

- **filter()**:
  - Returns a new array containing all elements that pass the test
  - Takes a callback function that returns true/false
  - If no elements match, returns an empty array
- **find()**:
  - Returns the first element that passes the test
  - Takes a callback function that returns true/false
  - If no element matches, returns undefined
- **findIndex()**:
  - Returns the index of the first element that passes the test

    - Takes a callback function that returns true/false
    - If no element matches, returns -1

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

// filter - returns all matching elements
const evens = numbers.filter(num => num % 2 === 0);
console.log(evens); // [2, 4]

// find - returns first matching element
const firstEven = numbers.find(num => num % 2 === 0);
console.log(firstEven); // 2

// findIndex - returns index of first matching element
const firstEvenIndex = numbers.findIndex(num => num % 2 === 0);
console.log(firstEvenIndex); // 1
```

66. How do reduce() and reduceRight() work?

- **reduce()**:
    - Executes a reducer function on each element, resulting in a single output value
    - Processes the array from left to right
    - Takes a callback with accumulator, current value, index, and array parameters
    - Also takes an initial value for the accumulator
- **reduceRight()**:
    - Same as reduce() but processes the array from right to left

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

// reduce - sum all numbers
const sum = numbers.reduce((accumulator, current) => {
  return accumulator + current;
}, 0); // 0 is the initial value
console.log(sum); // 10

// reduce - create an object from an array
const people = [
  { id: 1, name: "John" },
  { id: 2, name: "Jane" }
];

const peopleMap = people.reduce((acc, person) => {
  acc[person.id] = person;
  return acc;
}, {});
console.log(peopleMap); // { 1: {id: 1, name: "John"}, 2: {id: 2, name:
"Jane"} }

// reduceRight - concatenate from right to left
const letters = ["a", "b", "c"];
const reversed = letters.reduceRight((acc, letter) => acc + letter, "");
console.log(reversed); // "cba"
```

67. How do you check if something is an array?

There are several ways to check if a value is an array:

1. **Array.isArray()** (preferred, ES5+):
2. `Array.isArray([1, 2, 3]); // true`
3. `Array.isArray("string"); // false`
4. **instanceof operator** (less reliable due to cross-frame issues):
5. `[1, 2, 3] instanceof Array; // true`
6. **Object.prototype.toString**:
7. `Object.prototype.toString.call([1, 2, 3]) === '[object Array]'; // true`
8. **constructor property** (unreliable if modified):
9. `[1, 2, 3].constructor === Array; // true`

68. What is array destructuring?

Array destructuring is a JavaScript expression that allows you to extract values from arrays and assign them to variables in a single statement.

```
// Basic destructuring
const numbers = [1, 2, 3, 4, 5];
const [first, second, ...rest] = numbers;
console.log(first); // 1
console.log(second); // 2
console.log(rest); // [3, 4, 5]

// Skipping elements
const [a, , c] = numbers;
console.log(a); // 1
console.log(c); // 3

// Default values
const [x = 0, y = 0, z = 0] = [1, 2];
console.log(z); // 0 (default value)

// Swapping variables
let m = 1, n = 2;
[m, n] = [n, m];
console.log(m); // 2
console.log(n); // 1

// Nested array destructuring
const nested = [1, [2, 3], 4];
const [p, [q, r], s] = nested;
console.log(q); // 2
```

69. How do you remove elements from an array?

There are several ways to remove elements from an array:

1. **Remove the last element** with `pop()`:
2. `const arr = [1, 2, 3];`
3. `arr.pop(); // returns 3`
4. `console.log(arr); // [1, 2]`
5. **Remove the first element** with `shift()`:
6. `const arr = [1, 2, 3];`
7. `arr.shift(); // returns 1`
8. `console.log(arr); // [2, 3]`
9. **Remove elements by index** with `splice()`:

```
10.  const arr = [1, 2, 3, 4, 5];
11.  arr.splice(2, 1); // remove 1 element at index 2
12.  console.log(arr); // [1, 2, 4, 5]
```
13. **Filter out elements** (creates a new array):
```
14.  const arr = [1, 2, 3, 4, 5];
15.  const filtered = arr.filter(item => item !== 3);
16.  console.log(filtered); // [1, 2, 4, 5]
```
17. **Remove multiple elements by value**:
```
18.  const arr = [1, 2, 3, 2, 5];
19.  const removeValue = 2;
20.  const filtered = arr.filter(item => item !== removeValue);
21.  console.log(filtered); // [1, 3, 5]
```

70. How do you add elements to an array?

There are several ways to add elements to an array:

1. **Add to the end** with `push()`:
```
2. const arr = [1, 2, 3];
3. arr.push(4, 5); // returns new length 5
4. console.log(arr); // [1, 2, 3, 4, 5]
```
5. **Add to the beginning** with `unshift()`:
```
6. const arr = [1, 2, 3];
7. arr.unshift(-1, 0); // returns new length 5
8. console.log(arr); // [-1, 0, 1, 2, 3]
```
9. **Add at specific position** with `splice()`:
```
10.  const arr = [1, 2, 5];
11.  arr.splice(2, 0, 3, 4); // insert at index 2, delete 0 elements
12.  console.log(arr); // [1, 2, 3, 4, 5]
```
13. **Concatenate arrays** with `concat()` (creates a new array):
```
14.  const arr1 = [1, 2];
15.  const arr2 = [3, 4];
16.  const newArr = arr1.concat(arr2);
17.  console.log(newArr); // [1, 2, 3, 4]
```
18. **Using spread operator** (creates a new array):
```
19.  const arr = [1, 2];
20.  const newArr = [...arr, 3, 4];
21.  console.log(newArr); // [1, 2, 3, 4]
```

71. What is the difference between slice() and splice()?

- **slice()**:
  - Does NOT modify the original array
  - Returns a new array with a portion of the original array
  - Takes start and end parameters (end is exclusive)
  - Negative indices count from the end of the array
- **splice()**:
  - Modifies the original array
  - Returns an array containing the deleted elements
  - Takes start index, delete count, and items to add
  - Can remove elements, add elements, or both

```
// slice() - non-mutating
const arr = [1, 2, 3, 4, 5];
const sliced = arr.slice(1, 3);
console.log(sliced); // [2, 3]
console.log(arr); // [1, 2, 3, 4, 5] (unchanged)

// splice() - mutating
const arr2 = [1, 2, 3, 4, 5];
const removed = arr2.splice(1, 2, 'a', 'b'); // start at 1, remove 2, add
'a' and 'b'
console.log(removed); // [2, 3] (removed elements)
console.log(arr2); // [1, 'a', 'b', 4, 5] (modified)
```

72. How do you flatten arrays?

There are several ways to flatten nested arrays:

1. **flat()** method (ES2019):
```
2. const nested = [1, [2, [3, 4]]];
3. const flattened = nested.flat(); // one level deep by default
4. console.log(flattened); // [1, 2, [3, 4]]
5.
6. const deepFlattened = nested.flat(2); // specify depth
7. console.log(deepFlattened); // [1, 2, 3, 4]
8.
9. const infiniteFlattened = nested.flat(Infinity); // flatten all
   levels
10.  console.log(infiniteFlattened); // [1, 2, 3, 4]
```
11. **reduce() and concat()** (pre-ES2019):
```
12.  function flatten(arr) {
13.    return arr.reduce((acc, val) => {
14.      return acc.concat(Array.isArray(val) ? flatten(val) : val);
15.    }, []);
16.  }
17.
18.  const nested = [1, [2, [3, 4]]];
19.  console.log(flatten(nested)); // [1, 2, 3, 4]
```
20. **Using apply and concat** (pre-ES2019, one level):
```
21.  const nested = [1, [2, 3], [4]];
22.  const flattened = [].concat.apply([], nested);
23.  console.log(flattened); // [1, 2, 3, 4]
```

73. How do you sort arrays?

The `sort()` method sorts the elements of an array in place and returns the sorted array. By default, it converts elements to strings and compares their UTF-16 code units values.

```
// Default sort (lexicographic)
const fruits = ["banana", "apple", "orange"];
fruits.sort();
console.log(fruits); // ["apple", "banana", "orange"]

// Warning: numbers are converted to strings by default
const numbers = [10, 2, 5, 1, 30];
numbers.sort();
console.log(numbers); // [1, 10, 2, 30, 5] (incorrect numeric sort)

// Numeric sort with compare function
numbers.sort((a, b) => a - b); // ascending
```

```
console.log(numbers); // [1, 2, 5, 10, 30]

numbers.sort((a, b) => b - a); // descending
console.log(numbers); // [30, 10, 5, 2, 1]

// Sorting objects
const people = [
  { name: "John", age: 30 },
  { name: "Jane", age: 25 },
  { name: "Jim", age: 35 }
];

// Sort by age
people.sort((a, b) => a.age - b.age);
console.log(people.map(p => p.name)); // ["Jane", "John", "Jim"]

// Sort by name
people.sort((a, b) => a.name.localeCompare(b.name));
console.log(people.map(p => p.name)); // ["Jane", "Jim", "John"]
```

74. What are sparse arrays?

Sparse arrays are arrays that have "holes" in them - indices that don't have a value. They're created when you don't assign values to all indices consecutively.

```
// Creating sparse arrays
const sparse = new Array(3); // [empty × 3]
console.log(sparse.length); // 3
console.log(sparse[0]); // undefined

const anotherSparse = [];
anotherSparse[0] = "a";
anotherSparse[2] = "c";
console.log(anotherSparse); // ['a', empty, 'c']
console.log(anotherSparse.length); // 3
console.log(anotherSparse[1]); // undefined

// Behavior with array methods
sparse.forEach(item => console.log(item)); // Nothing happens (skips empty
slots)
console.log(sparse.map(item => item)); // [empty × 3] (preserves holes)

// Convert sparse to dense
const dense = [...sparse].fill(undefined);
console.log(dense); // [undefined, undefined, undefined]
```

75. How do some() and every() work?

- **some()**:
  - Tests whether at least one element in the array passes the test
  - Returns true if at least one element passes, otherwise false
  - Returns false for empty arrays
- **every()**:
  - Tests whether all elements in the array pass the test
  - Returns true if all elements pass, otherwise false
  - Returns true for empty arrays

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

// some() - at least one element passes
const hasEven = numbers.some(num => num % 2 === 0);
console.log(hasEven); // true (2 and 4 are even)

const hasNegative = numbers.some(num => num < 0);
console.log(hasNegative); // false (no negatives)

// every() - all elements pass
const allPositive = numbers.every(num => num > 0);
console.log(allPositive); // true (all are positive)

const allEven = numbers.every(num => num % 2 === 0);
console.log(allEven); // false (not all are even)
```

76. What is the spread operator and how is it used with arrays?

The spread operator (`...`) allows an iterable (like an array) to be expanded in places where zero or more arguments or elements are expected.

Common uses with arrays:

1. **Copying arrays**:
2. `const original = [1, 2, 3];`
3. `const copy = [...original];`
4. **Concatenating arrays**:
5. `const arr1 = [1, 2];`
6. `const arr2 = [3, 4];`
7. `const combined = [...arr1, ...arr2]; // [1, 2, 3, 4]`
8. **Spreading elements as arguments**:
9. `const numbers = [1, 2, 3];`
10. `const max = Math.max(...numbers); // Same as Math.max(1, 2, 3)`
11. **Adding elements to an array**:
12. `const arr = [2, 3];`
13. `const newArr = [1, ...arr, 4]; // [1, 2, 3, 4]`
14. **Converting iterables to arrays**:
15. `const str = "hello";`
16. `const chars = [...str]; // ['h', 'e', 'l', 'l', 'o']`

77. How do you convert an array-like object to an array?

Array-like objects have a length property and indexed elements but don't have array methods. Examples include the `arguments` object, DOM NodeList, and strings.

Ways to convert array-like objects to arrays:

1. **Using Array.from()** (ES6+):
2. `function example() {`
3. `  const args = Array.from(arguments);`
4. `  args.forEach(arg => console.log(arg));`
5. `}`
6.

```
7. const nodeList = document.querySelectorAll('div');
8. const divArray = Array.from(nodeList);
```
9. **Using spread operator** (ES6+):
```
10.  function example() {
11.    const args = [...arguments];
12.    args.forEach(arg => console.log(arg));
13.  }
14.
15.  const nodeList = document.querySelectorAll('div');
16.  const divArray = [...nodeList];
```
17. **Using Array.prototype.slice.call()** (pre-ES6):
```
18.  function example() {
19.    const args = Array.prototype.slice.call(arguments);
20.    args.forEach(arg => console.log(arg));
21.  }
22.
23.  const nodeList = document.querySelectorAll('div');
24.  const divArray = Array.prototype.slice.call(nodeList);
```

78. What is the difference between for...in and for...of loops?

The difference between for...in and for...of loops lies in what they iterate over:

- **for...in loop**: Iterates over all enumerable property names (keys) of an object. When used with arrays, it iterates over the array indices (as strings), not the values. It's primarily designed for objects.
- **for...of loop**: Iterates over the values of iterable objects like arrays, strings, maps, sets, etc. It provides direct access to the values rather than their indices or property names.

Example:

```
// for...in with an array
const array = ['a', 'b', 'c'];
for (let index in array) {
  console.log(index); // Outputs: "0", "1", "2" (as strings)
}

// for...of with an array
for (let value of array) {
  console.log(value); // Outputs: "a", "b", "c"
}
```

79. How do you compare arrays?

There is no built-in method to directly compare arrays for equality. Arrays are objects in JavaScript, so comparing them with == or === only checks if they reference the same object in memory.

To compare arrays for equality:

1. **Check if they have the same length**
2. **Compare each element at corresponding positions**

Example:

```
function areArraysEqual(arr1, arr2) {
  if (arr1.length !== arr2.length) return false;

  for (let i = 0; i < arr1.length; i++) {
    if (arr1[i] !== arr2[i]) return false;
  }

  return true;
}
```

For nested arrays or complex objects within arrays, you'd need a deeper comparison function.

80. How do you remove duplicate values from an array?

Several methods can be used:

1. **Using Set** (most common modern approach):

```
const array = [1, 2, 2, 3, 4, 4, 5];
const uniqueArray = [...new Set(array)];
// Result: [1, 2, 3, 4, 5]
```

2. **Using filter() and indexOf()**:

```
const array = [1, 2, 2, 3, 4, 4, 5];
const uniqueArray = array.filter((item, index) => {
  return array.indexOf(item) === index;
});
```

3. **Using reduce()**:

```
const array = [1, 2, 2, 3, 4, 4, 5];
const uniqueArray = array.reduce((unique, item) => {
  return unique.includes(item) ? unique : [...unique, item];
}, []);
```

DOM Manipulation

81. What is the DOM?

The DOM (Document Object Model) is a programming interface for web documents. It represents the structure of HTML and XML documents as a tree of objects that can be manipulated with JavaScript. The DOM provides a way to:

- Access and modify the content, structure, and style of a document
- React to user interactions and events
- Create dynamic, interactive web pages

The DOM represents the document as nodes and objects. Each element, attribute, and piece of text in the HTML becomes a DOM node in the tree structure, which can then be accessed, modified, added, or removed using JavaScript.

82. How do you select elements in the DOM?

There are several methods to select elements in the DOM:

1. **getElementById()**: Selects a single element by its ID attribute

   ```
   const element = document.getElementById('myId');
   ```

2. **getElementsByClassName()**: Returns a live HTMLCollection of elements with the specified class

   ```
   const elements = document.getElementsByClassName('myClass');
   ```

3. **getElementsByTagName()**: Returns a live HTMLCollection of elements with the specified tag name

   ```
   const paragraphs = document.getElementsByTagName('p');
   ```

4. **querySelector()**: Returns the first element that matches a CSS selector

   ```
   const element = document.querySelector('.myClass');
   ```

5. **querySelectorAll()**: Returns a static NodeList of all elements matching a CSS selector

```
const elements = document.querySelectorAll('div.container > p');
```

83. What is the difference between querySelector() and getElementById()?

The main differences between querySelector() and getElementById() are:

1. **Selector Syntax**:
   - getElementById() accepts only a string ID (without the # prefix)
   - querySelector() accepts any valid CSS selector (classes, IDs with #, tags, attributes, pseudo-classes, etc.)
2. **Return Value**:
   - getElementById() returns null if no matching element is found
   - querySelector() also returns null if no match is found
3. **Performance**:
   - getElementById() is generally faster since it's specifically optimized for ID lookup
   - querySelector() is more versatile but slightly slower as it needs to parse and match the CSS selector
4. **Specificity**:
   - getElementById() can only find elements by ID
   - querySelector() can find elements using any CSS selector pattern

Example:

```
// These do the same thing:
const element1 = document.getElementById('myId');
const element2 = document.querySelector('#myId');

// But querySelector can do more:
const firstParagraph = document.querySelector('p');
const activeButton = document.querySelector('button.active');
```

84. How do you create and append elements to the DOM?

To create and append elements to the DOM:

1. **Create an element** using `document.createElement()`
2. **Set properties** on the element (content, attributes, classes, etc.)
3. **Append the element** to an existing DOM element

Example:

```javascript
// Create a new element
const newParagraph = document.createElement('p');

// Set content and attributes
newParagraph.textContent = 'This is a new paragraph.';
newParagraph.className = 'info-text';
newParagraph.id = 'newPara';

// Append to an existing element
document.body.appendChild(newParagraph);

// Alternative ways to append (newer methods)
parentElement.append(newParagraph); // Can append multiple nodes and text
parentElement.prepend(newParagraph); // Adds at beginning of parent
existingElement.before(newParagraph); // Adds before an existing element
existingElement.after(newParagraph); // Adds after an existing element
```

85. How do you modify element attributes?

You can modify element attributes in several ways:

1. **Using the direct property** (for standard attributes):

   ```javascript
   element.id = 'newId';
   element.className = 'new-class';
   element.href = 'https://example.com';
   ```

2. **Using getAttribute() and setAttribute()**:

   ```javascript
   // Get attribute
   const currentSrc = element.getAttribute('src');

   // Set attribute
   element.setAttribute('data-value', '123');
   element.setAttribute('disabled', '');
   ```

3. **Removing attributes** with removeAttribute():

   ```javascript
   element.removeAttribute('disabled');
   ```

4. **Handling data attributes**:

   ```javascript
   // Using dataset property
   ```

```
element.dataset.userId = '42';  // Sets data-user-id="42"
const userId = element.dataset.userId;  // Gets value of data-user-id
```

5. **Checking attributes**:

```
Example
if (element.hasAttribute('required')) {
  // Do something
}
```

86. What is the difference between innerHTML, textContent, and innerText?

innerHTML:

- Sets or gets the HTML content within an element
- Parses content as HTML, so it can include HTML tags that will render as elements
- Can pose security risks if used with user-provided content (potential XSS attacks)
- Changes trigger reflow and repaint as the DOM structure is modified

textContent:

- Sets or gets the text content of an element and all its descendants
- Returns all text content regardless of CSS styling or visibility
- Does not parse HTML tags (treats them as literal text)
- Generally more performant than innerHTML when just dealing with text

innerText:

- Returns the visible text content of an element (respects CSS styling)
- Aware of styling and won't return text that is hidden with CSS
- Triggers reflow as it needs to calculate styles to determine visibility
- Also doesn't parse HTML tags (treats them as literal text)

Example:

```
// Given this HTML: <div id="example"><p
style="display:none">Hidden</p><p>Visible</p></div>

const div = document.getElementById('example');
div.innerHTML; // "<p style="display:none">Hidden</p><p>Visible</p>"
div.textContent; // "HiddenVisible" (includes hidden text)
div.innerText; // "Visible" (only includes visible text)
```

87. How do you handle events in JavaScript?

There are multiple ways to handle events in JavaScript:

1. HTML Event Attributes (not recommended):

```
<button onclick="alert('Clicked!')">Click me</button>
```

2. DOM Element Properties:

```
const button = document.getElementById('myButton');
button.onclick = function() {
  console.log('Button clicked!');
};
```

3. addEventListener() (recommended):

```
const button = document.getElementById('myButton');
button.addEventListener('click', function(event) {
  console.log('Button clicked!');
  console.log(event); // Access the event object
});
```

Key advantages of `addEventListener()`:

- Can add multiple event listeners to a single element
- More control over event propagation
- Ability to remove event listeners with `removeEventListener()`
- Provides the event object with useful information

4. Event delegation (for efficiency with many elements):

```
document.getElementById('parent-list').addEventListener('click',
function(event) {
  if (event.target.tagName === 'LI') {
    console.log('List item clicked:', event.target.textContent);
  }
});
```

88. What is event propagation?

Event propagation describes how events travel through the DOM tree. It occurs in three phases:

1. Capturing Phase:

- The event starts at the root of the document and travels down to the target element
- Rarely used, but can be captured by setting the third parameter of addEventListener to true

2. Target Phase:

- The event reaches the target element that triggered it

3. Bubbling Phase:

- The event bubbles up from the target element back to the root of the document
- This is the default behavior that most event handlers respond to

Example:

```
// Capturing phase (third parameter is true)
parent.addEventListener('click', function() {
  console.log('Parent clicked - capturing phase');
}, true);

// Bubbling phase (default - third parameter is false/omitted)
child.addEventListener('click', function() {
  console.log('Child clicked - bubbling phase');
});
```

When the child element is clicked, events fire in this order:

1. Parent (capturing)
2. Child (target)
3. Parent (bubbling)

89. How do you prevent default behavior in events?

The `preventDefault()` method is used to stop the default action of an element from occurring. Common examples include:

- Preventing form submission
- Stopping link navigation
- Preventing checkbox state changes

Example:

```
// Prevent form submission
document.getElementById('myForm').addEventListener('submit',
function(event) {
  event.preventDefault();
  // Custom validation or AJAX submission instead
  console.log('Form submission prevented');
});

// Prevent link navigation
document.querySelector('a').addEventListener('click', function(event) {
  event.preventDefault();
  console.log('Link click prevented');
});
```

90. How do you stop event propagation?

To stop event propagation (prevent the event from bubbling up or capturing down), use the `stopPropagation()` method on the event object:

```
childElement.addEventListener('click', function(event) {
  event.stopPropagation();
  console.log('Child clicked, but event won\'t bubble to parent');
});

parentElement.addEventListener('click', function() {
  console.log('This won\'t run when child is clicked');
});
```

For an even stronger version that prevents any other listeners on the same element from firing, use `stopImmediatePropagation()`:

```
element.addEventListener('click', function(event) {
  event.stopImmediatePropagation();
  console.log('This runs');
});

element.addEventListener('click', function() {
  console.log('This will NOT run');
});
```

91. What is event delegation and why is it useful?

Event delegation is a technique of attaching a single event listener to a parent element instead of attaching individual listeners to multiple child elements.

How it works:

1. Attach one event listener to a parent element
2. When the event occurs on a child element, it bubbles up to the parent
3. Check the event target to determine which child initiated the event
4. Execute the appropriate code based on the target

Benefits:

- **Memory efficiency**: Reduces the number of event listeners
- **Dynamic elements**: Works with elements added or removed after page load
- **Less code**: Simplifies maintenance with fewer event bindings
- **Cleaner markup**: Reduces inline event handlers

Example:

```
// Instead of adding click events to each button
document.getElementById('button-container').addEventListener('click',
function(event) {
  // Check if the clicked element is a button
  if (event.target.tagName === 'BUTTON') {
    console.log('Button clicked:', event.target.textContent);

    // Use data attributes for specific actions
    if (event.target.dataset.action === 'delete') {
      deleteItem(event.target.dataset.id);
    }
  }
});
```

92. How do you dynamically add and remove CSS classes?

Modern browsers provide the `classList` API for manipulating CSS classes:

Adding classes:

```
element.classList.add('active');
element.classList.add('highlight', 'visible'); // Add multiple classes
```

Removing classes:

```
element.classList.remove('active');
element.classList.remove('highlight', 'visible'); // Remove multiple
classes
```

Toggling classes (adds if absent, removes if present):

```
element.classList.toggle('active'); // Returns boolean indicating new state
element.classList.toggle('visible', isVisible); // Force add/remove based
on second parameter
```

Checking if an element has a class:

```
if (element.classList.contains('active')) {
  // Do something
}
```

Replacing a class:

```
element.classList.replace('old-class', 'new-class');
```

Older fallback method (less recommended):

```
// Add class
element.className += ' new-class';

// Remove class
element.className = element.className.replace(/\bclass-name\b/, '');
```

93. How do you manipulate CSS styles using JavaScript?

CSS styles can be manipulated in JavaScript using the `style` property:

Setting individual styles:

```
element.style.color = 'blue';
element.style.backgroundColor = 'yellow'; // Note camelCase for hyphenated
CSS properties
element.style.marginTop = '20px';
```

Setting multiple styles at once:

```
Object.assign(element.style, {
  color: 'white',
  backgroundColor: 'black',
  padding: '10px'
});
```

Accessing computed styles (including those set in CSS files):

```
const computedStyle = window.getComputedStyle(element);
const currentColor = computedStyle.color;
const currentWidth = computedStyle.width;
```

Setting inline styles with cssText (overwrites all existing inline styles):

```
element.style.cssText = 'color: red; background-color: white; font-size:
16px;';
```

Adding important styles:

```
element.style.setProperty('color', 'red', 'important');
```

Removing styles:

```
element.style.removeProperty('background-color');
```

94. What is the difference between window, document, and screen?

window:

- The global object in browser JavaScript representing the browser window/tab
- The top-level object in the browser object hierarchy
- Contains properties like location, history, navigator, localStorage
- Provides methods like alert(), setTimeout(), fetch()
- Can reference global variables and functions directly

document:

- A property of the window object: `window.document`
- Represents the HTML document loaded in the window
- Entry point to the DOM (Document Object Model)
- Provides methods like getElementById(), querySelector(), createElement()
- Contains the page content structure (HTML elements)

screen:

- A property of the window object: `window.screen`
- Represents the user's physical screen/monitor
- Provides information about the user's display capabilities
- Contains properties like screen.width, screen.height, screen.availWidth, screen.availHeight, screen.colorDepth
- Useful for responsive design and detecting device capabilities

Example:

```
// Window methods and properties
window.innerHeight; // Height of browser viewport
window.open('https://example.com'); // Open a new window

// Document methods and properties
document.title = 'New Page Title';
document.body.style.backgroundColor = 'lightblue';

// Screen properties
const screenWidth = screen.width;
const screenHeight = screen.height;
```

95. How do you create a custom event?

Creating and dispatching custom events involves three main steps:

1. Create the event using either:

- `new Event()` for simple events
- `new CustomEvent()` for events with custom data

2. Add event listeners to respond to the custom event

3. Dispatch the event using `element.dispatchEvent()`

Example with `Event`:

```
// Create a simple event
const simpleEvent = new Event('simpleEvent', {
  bubbles: true,      // Whether the event bubbles up through the DOM
  cancelable: true    // Whether the event can be canceled
});

// Add a listener for the event
document.addEventListener('simpleEvent', function(e) {
  console.log('Simple event triggered');
});

// Dispatch the event
document.dispatchEvent(simpleEvent);
```

Example with `CustomEvent` (includes custom data):

```
// Create an event with custom data
const productEvent = new CustomEvent('productAdded', {
  bubbles: true,
  cancelable: true,
  detail: {              // Custom data passed with the event
    productId: 123,
    productName: 'Keyboard',
    price: 59.99
  }
});

// Add a listener that uses the custom data
document.addEventListener('productAdded', function(e) {
  console.log('Product added:', e.detail.productName, 'for $' +
e.detail.price);
});

// Dispatch the event
document.dispatchEvent(productEvent);
```

96. What is the Virtual DOM?

The Virtual DOM is a programming concept where an ideal, or "virtual", representation of a UI is kept in memory and synced with the "real" DOM by a library such as React. This process is called reconciliation.

Key characteristics:

- **Lightweight JavaScript representation** of the actual DOM
- **In-memory structure** that tracks changes to the UI
- **Abstracts manual DOM manipulation** to improve performance and developer experience

How it works:

1. When state changes in an application, a new virtual DOM tree is created
2. This new virtual DOM is compared with the previous virtual DOM (diffing)
3. Only the changed elements are updated in the real DOM (patching)
4. This minimizes direct DOM manipulations, which are slow and expensive

Benefits:

- **Performance**: Batch DOM updates for efficiency
- **Abstraction**: Developers think declaratively rather than imperatively
- **Cross-platform**: Virtual DOM can render to platforms other than browsers
- **Testability**: Components can be tested without a browser

The Virtual DOM is most prominently used in React but is also implemented in other frameworks and libraries.

97. How do you detect clicks outside of an element?

To detect clicks outside of a specific element:

1. Add a click event listener to the document or window
2. Check if the clicked element is within the target element using `contains()`
3. If not, it's a click outside the element

Example:

```
// Element we want to detect outside clicks for
const menu = document.getElementById('dropdown-menu');
const button = document.getElementById('menu-button');

// Add click listener to the document
document.addEventListener('click', function(event) {
  const isClickInside = menu.contains(event.target) ||
button.contains(event.target);

  if (!isClickInside) {
    // Click happened outside the menu and button
    menu.classList.remove('show');
    console.log('Clicked outside, closing menu');
  }
});

// Toggle menu when button is clicked
button.addEventListener('click', function() {
  menu.classList.toggle('show');
});
```

For React components, a similar approach can be implemented using refs and useEffect:

```
function Dropdown() {
  const [isOpen, setIsOpen] = useState(false);
  const dropdownRef = useRef(null);

  useEffect(() => {
    function handleClickOutside(event) {
      if (dropdownRef.current &&
!dropdownRef.current.contains(event.target)) {
        setIsOpen(false);
      }
    }

    document.addEventListener('mousedown', handleClickOutside);
    return () => {
      document.removeEventListener('mousedown', handleClickOutside);
    };
  }, []);

  return (
    <div ref={dropdownRef}>
      <button onClick={() => setIsOpen(!isOpen)}>Toggle</button>
      {isOpen && <div className="dropdown-menu">Menu content</div>}
    </div>
  );
}
```

98. How do you implement drag and drop functionality?

Implementing drag and drop functionality can be done using the HTML5 Drag and Drop API:

1. Make elements draggable:

```
<div id="draggable" draggable="true">Drag me</div>
```

2. Add drag event listeners to the draggable element:

```
const draggable = document.getElementById('draggable');

draggable.addEventListener('dragstart', function(event) {
  // Store data being dragged
  event.dataTransfer.setData('text/plain', event.target.id);

  // Optional: Change appearance during drag
  event.target.classList.add('dragging');
});

draggable.addEventListener('dragend', function(event) {
  // Remove appearance changes
  event.target.classList.remove('dragging');
});
```

3. Set up drop zones:

```
const dropZone = document.getElementById('drop-zone');

// Prevent default to allow drop
dropZone.addEventListener('dragover', function(event) {
  event.preventDefault();
});

// Optional: Visual feedback when dragging over the zone
dropZone.addEventListener('dragenter', function(event) {
  event.preventDefault();
  this.classList.add('drag-over');
});

dropZone.addEventListener('dragleave', function() {
  this.classList.remove('drag-over');
});

// Handle the drop
dropZone.addEventListener('drop', function(event) {
  event.preventDefault();

  // Remove visual feedback
  this.classList.remove('drag-over');

  // Get the dragged element's ID from dataTransfer
  const id = event.dataTransfer.getData('text/plain');
  const draggable = document.getElementById(id);

  // Append to the drop zone
  this.appendChild(draggable);

  console.log('Dropped element:', draggable);
});
```

4. Optional: Add custom drag image:

```
draggable.addEventListener('dragstart', function(event) {
  // Create custom drag image
  const dragImage = document.createElement('div');
  dragImage.textContent = 'Dragging: ' + this.textContent;
  dragImage.style.cssText = 'position: absolute; opacity: 0.8; z-index: -
1;';
  document.body.appendChild(dragImage);

  // Set the custom drag image
  event.dataTransfer.setDragImage(dragImage, 0, 0);
  // Store data
  event.dataTransfer.setData('text/plain', this.id);

  // Clean up the temporary element after a short delay
  setTimeout(() => {
    document.body.removeChild(dragImage);
  }, 0);
});
```

For more complex drag and drop functionality, libraries like SortableJS or react-beautiful-dnd provide higher-level abstractions.

99. What are data attributes and how do you access them?

Data attributes are custom attributes that allow you to store extra information on HTML elements without using non-standard attributes, extra properties on DOM elements, or Node.setUserData().

1. Adding data attributes in HTML:

```
<div id="user"
     data-id="123"
     data-user-name="johndoe"
     data-role="admin"
     data-last-login="2023-04-15">
  John Doe
</div>
```

2. Accessing data attributes in JavaScript:

Using the `dataset` property (modern, preferred method):

```
const userElement = document.getElementById('user');

// Reading values
const userId = userElement.dataset.id; // "123"
const userName = userElement.dataset.userName; // "johndoe" (note camelCase
conversion)
const userRole = userElement.dataset.role; // "admin"

// Writing values
userElement.dataset.status = "active";
userElement.dataset.lastLogin = "2023-05-01";
```

Using getAttribute/setAttribute (alternative method):

```
// Reading
const userId = userElement.getAttribute('data-id');
const userName = userElement.getAttribute('data-user-name');

// Writing
userElement.setAttribute('data-status', 'active');
```

3. Accessing data attributes in CSS:

```
/* Select elements with specific data attribute */
[data-role="admin"] {
  background-color: #ffecec;
}

/* Style based on data attribute value */
div::before {
  content: attr(data-user-name);
  font-weight: bold;
}
```

Key points about data attributes:

- Names are converted from kebab-case to camelCase when accessed via dataset
- All values are stored as strings
- Provide a clean way to associate data with elements without cluttering the DOM
- Particularly useful for JavaScript frameworks and custom components
- Should not be used for data that should be visible to users (use appropriate HTML elements instead)

100. How do you measure element dimensions and position?

JavaScript provides several methods to measure element dimensions and position:

- **Element dimensions**:
  - `element.offsetWidth/offsetHeight`: Width/height including padding and border
  - `element.clientWidth/clientHeight`: Width/height including padding but not border
  - `element.scrollWidth/scrollHeight`: Complete content dimensions including overflow
- **Element position**:
  - `element.getBoundingClientRect()`: Returns a DOMRect object with properties like top, left, right, bottom, width, and height relative to the viewport
  - `element.offsetLeft/offsetTop`: Position relative to the offsetParent
  - For position relative to the document: Add scroll position to getBoundingClientRect values
  - `getComputedStyle(element)`: For getting specific CSS-computed dimensions

Asynchronous JavaScript

101. What are Promises in JavaScript?

Promises are objects representing the eventual completion or failure of an asynchronous operation and its resulting value. A Promise can be in one of three states: pending, fulfilled, or rejected. They provide a clean way to handle asynchronous operations without callback hell.

102. How do you create a Promise?

You create a Promise using the Promise constructor, which takes an executor function with two parameters: resolve and reject.

```
const myPromise = new Promise((resolve, reject) => {
  // Asynchronous operation
  if (/* operation successful */) {
    resolve(value); // Fulfills the promise
  } else {
    reject(error); // Rejects the promise
  }
});
```

103. What are the states of a Promise?

A Promise can be in one of three states:

- **Pending**: Initial state, neither fulfilled nor rejected
- **Fulfilled**: The operation completed successfully
- **Rejected**: The operation failed Once a promise is fulfilled or rejected, it is settled and cannot change to another state.

104. What is Promise chaining?

Promise chaining is a technique that allows you to execute multiple asynchronous operations in sequence, where each subsequent operation starts when the previous operation succeeds. It's done by using the `then()` method which returns a new Promise.

```
fetchData()
  .then(data => processData(data))
  .then(processedData => displayData(processedData))
  .catch(error => handleError(error));
```

105. How do you handle errors in Promises?

Error handling in Promises can be done using:

- The `catch()` method at the end of a Promise chain
- The second argument of the `then()` method
- Using `try/catch` blocks with async/await

```
myPromise
  .then(handleSuccess)
  .catch(handleError);

// OR
myPromise.then(
  handleSuccess,
  handleError
);
```

106. What is Promise.all() and when would you use it?

`Promise.all()` takes an iterable of Promises and returns a single Promise that resolves when all input promises have resolved, or rejects when any of the input promises rejects. It's useful when you need to run multiple asynchronous operations in parallel and wait for all of them to complete.

```
Promise.all([fetchUser(), fetchPosts(), fetchComments()])
  .then(([user, posts, comments]) => {
    // All three promises resolved
  })
  .catch(error => {
    // At least one promise rejected
  });
```

107. What is Promise.race() and when would you use it?

`Promise.race()` takes an iterable of Promises and returns a Promise that resolves or rejects as soon as one of the promises in the iterable resolves or rejects. It's useful for implementing timeouts or when you want to use the result of the first completed operation.

```
Promise.race([
  fetch('/resource'),
  new Promise((_, reject) => setTimeout(() => reject(new Error('Timeout')),
5000))
])
.then(response => console.log(response))
.catch(error => console.error(error));
```

108. What is Promise.allSettled()?

`Promise.allSettled()` takes an iterable of Promises and returns a Promise that resolves after all of the given promises have either fulfilled or rejected. Unlike `Promise.all()`, it will never reject. The returned promise resolves with an array of objects describing the outcome of each promise.

```
Promise.allSettled([fetchUser(), fetchPosts(), fetchComments()])
  .then(results => {
    results.forEach(result => {
      if (result.status === 'fulfilled') {
        console.log('Value:', result.value);
      } else {
        console.log('Reason:', result.reason);
      }
    });
  });
```

109. What is Promise.any()?

`Promise.any()` takes an iterable of Promises and returns a Promise that fulfills as soon as one of the promises fulfills, with the value of the fulfilled promise. If all promises are rejected, it rejects with an AggregateError containing all rejection reasons.

```
Promise.any([
  fetch('https://api.example.com/endpoint-1'),
  fetch('https://api.example.com/endpoint-2'),
  fetch('https://api.example.com/endpoint-3')
])
.then(firstResponse => console.log(firstResponse))
.catch(error => console.error(error));
```

110. What is the async/await syntax?

Async/await is a syntactic sugar on top of Promises, making asynchronous code look and behave more like synchronous code. An `async` function returns a Promise implicitly, and the `await` keyword can be used inside async functions to pause execution until the Promise resolves.

```
async function fetchUserData() {
  try {
    const response = await fetch('/api/user');
    const userData = await response.json();
    return userData;
  } catch (error) {
    console.error('Error fetching user data:', error);
  }
}
```

111. How do async/await and Promises relate to each other?

Async/await is built on top of Promises:

- `async` functions always return a Promise
- `await` can only be used inside `async` functions
- `await` expression pauses the execution until the Promise resolves or rejects
- Async/await makes Promise-based code more readable and easier to reason about

112. How do you handle errors in async/await?

Errors in async/await are handled using standard try/catch blocks:

```
async function fetchData() {
  try {
    const response = await fetch('/api/data');
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }
    const data = await response.json();
    return data;
  } catch (error) {
    console.error('Error fetching data:', error);
    // Handle error appropriately
  }
}
```

113. What is the Fetch API?

The Fetch API provides a modern interface for making HTTP requests. It returns Promises, making it a cleaner alternative to XMLHttpRequest. The `fetch()` function is the primary interface, which takes a URL and optional configuration object.

114. How do you make HTTP requests with Fetch?

```
// Basic GET request
fetch('https://api.example.com/data')
  .then(response => {
    if (!response.ok) {
      throw new Error(`HTTP error! Status: ${response.status}`);
    }
    return response.json();
  })
  .then(data => console.log(data))
  .catch(error => console.error('Error:', error));

// POST request with JSON data
fetch('https://api.example.com/create', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    name: 'John Doe',
    email: 'john@example.com'
  })
})
.then(response => response.json())
.then(data => console.log(data))
.catch(error => console.error('Error:', error));
```

115. What are AbortControllers?

AbortControllers provide a way to cancel fetch requests. They're part of the Fetch API and allow you to abort one or multiple fetch operations when needed.

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

fetch('https://api.example.com/data', { signal })
  .then(response => response.json())
  .then(data => console.log(data))
  .catch(error => {
    if (error.name === 'AbortError') {
      console.log('Fetch aborted');
    } else {
      console.error('Error:', error);
    }
  });

// Abort the fetch after 5 seconds
setTimeout(() => controller.abort(), 5000);
```

116. What is AJAX?

AJAX (Asynchronous JavaScript and XML) is a set of web development techniques using many web technologies on the client side to create asynchronous web applications. With AJAX, web applications can send and retrieve data from a server asynchronously without interfering with the display and behavior of the existing page.

117. How do you use XMLHttpRequest?

XMLHttpRequest (XHR) is the traditional way to make asynchronous HTTP requests before Fetch API:

```
const xhr = new XMLHttpRequest();
xhr.open('GET', 'https://api.example.com/data', true);

xhr.onload = function() {
  if (xhr.status >= 200 && xhr.status < 300) {
    const data = JSON.parse(xhr.responseText);
    console.log(data);
  } else {
    console.error('Request failed with status:', xhr.status);
  }
};

xhr.onerror = function() {
  console.error('Request failed');
};

xhr.send();
```

118. What are Web Workers?

Web Workers are a way to run JavaScript in background threads, separate from the main execution thread. They allow for multi-threading in JavaScript, enabling CPU-intensive tasks to run without blocking the UI.

```
// Main thread
const worker = new Worker('worker.js');

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

worker.postMessage({ data: 'some data' });

// In worker.js
self.onmessage = function(event) {
  const result = performExpensiveOperation(event.data);
  self.postMessage(result);
};
```

119. What is the Service Worker API?

Service Workers act as proxy servers that sit between web applications, the browser, and the network. They enable features like offline functionality, background sync, and push notifications by intercepting network requests and caching resources.

```
// Registering a service worker
navigator.serviceWorker.register('/service-worker.js')
  .then(registration => {
    console.log('Service Worker registered with scope:',
registration.scope);
  })
  .catch(error => {
    console.error('Service Worker registration failed:', error);
  });
```

120. How do microtasks and macrotasks differ in the event loop?

In the JavaScript event loop:

- **Macrotasks**: Include setTimeout, setInterval, setImmediate, I/O operations, UI rendering
- **Microtasks**: Include Promise callbacks, queueMicrotask, MutationObserver

The key difference is in their execution timing:

1. The event loop executes one macrotask
2. Then executes all microtasks in the microtask queue
3. Then performs UI rendering if needed
4. Then goes back to step 1

This means microtasks can "jump the queue" ahead of other macrotasks and are executed before the next macrotask.

ES6+ Features

121. What are template literals?

Template literals are string literals that allow embedded expressions and multi-line strings. They are enclosed by backticks ( ) instead of quotes.

```
const name = 'Alice';
const greeting = `Hello, ${name}!
Welcome to our website.`;
```

122. What are default parameters?

Default parameters allow function parameters to have default values if no value or undefined is passed:

```
function greet(name = 'Guest', greeting = 'Hello') {
  return `${greeting}, ${name}!`;
}

greet();            // "Hello, Guest!"
greet('Alice');     // "Hello, Alice!"
greet('Bob', 'Hi'); // "Hi, Bob!"
```

123. What is destructuring assignment?

Destructuring assignment is a syntax that allows you to extract values from arrays or properties from objects into distinct variables:

```
// Array destructuring
const [first, second, ...rest] = [1, 2, 3, 4, 5];
// first = 1, second = 2, rest = [3, 4, 5]

// Object destructuring
const { name, age, country = 'Unknown' } = { name: 'Alice', age: 30 };
// name = 'Alice', age = 30, country = 'Unknown'
```

124. What is the spread operator?

The spread operator (`...`) allows an iterable (like an array or string) to be expanded in places where zero or more arguments/elements are expected:

```
// Array spreading
const arr1 = [1, 2, 3];
const arr2 = [...arr1, 4, 5]; // [1, 2, 3, 4, 5]

// Object spreading
const obj1 = { x: 1, y: 2 };
const obj2 = { ...obj1, z: 3 }; // { x: 1, y: 2, z: 3 }

// In function calls
const numbers = [1, 2, 3];
Math.max(...numbers); // 3
```

125. What are rest parameters?

Rest parameters allow a function to accept an indefinite number of arguments as an array:

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

sum(1, 2, 3, 4); // 10
```

126. What are Symbols in JavaScript?

Symbols are a primitive data type introduced in ES6 that represent unique identifiers. Every Symbol value returned from Symbol() is unique, making them useful for object property keys to avoid name collisions.

```
const id = Symbol('id');
const user = {
  name: 'Alice',
  [id]: 12345 // Using a Symbol as a property key
};

// Symbols are not enumerable in for...in loops
for (let key in user) {
  console.log(key); // Only outputs "name"
}
```

127. What are Maps and Sets?

Map is a collection of keyed data items where keys can be of any type:

```
const userRoles = new Map();
userRoles.set('john', 'admin');
userRoles.set('jane', 'editor');
userRoles.get('john'); // "admin"
```

Set is a collection of unique values:

```
const uniqueNumbers = new Set([1, 2, 3, 3, 4, 4]);
// uniqueNumbers contains 1, 2, 3, 4
uniqueNumbers.add(5);
uniqueNumbers.has(3); // true
```

128. What are WeakMaps and WeakSets?

WeakMap is similar to Map, but its keys must be objects and are held "weakly", allowing them to be garbage-collected if there are no other references:

```
const userMetadata = new WeakMap();
let user = { name: 'John' };
userMetadata.set(user, { lastActive: Date.now() });
// If 'user' is later set to null, both the 'user' object
// and its associated metadata can be garbage-collected
```

WeakSet is similar to Set, but can only contain objects and holds them weakly:

```
const visited = new WeakSet();
let page1 = { url: '/home' };
visited.add(page1);
// If 'page1' is later set to null, it can be garbage-collected
```

129. What are iterators and iterables?

Iterables are objects implementing the Symbol.iterator method, allowing them to be iterated over (e.g., arrays, strings, Maps, Sets).

Iterators are objects with a next() method that returns { value, done } objects, used to iterate through a collection.

```
// Custom iterable
```

```
const range = {
  from: 1,
  to: 5,
  [Symbol.iterator]() {
    return {
      current: this.from,
      last: this.to,
      next() {
        if (this.current <= this.last) {
          return { value: this.current++, done: false };
        } else {
          return { done: true };
        }
      }
    };
  }
};

for (let num of range) {
  console.log(num); // 1, 2, 3, 4, 5
}
```

130. What is the for...of loop?

The for...of loop iterates over iterable objects (arrays, strings, Maps, Sets, etc.), invoking the object's iterator:

```
const array = ['a', 'b', 'c'];

for (const element of array) {
  console.log(element); // "a", "b", "c"
}

const string = "hello";
for (const char of string) {
  console.log(char); // "h", "e", "l", "l", "o"
}
```

131. What are Proxies?

Proxies provide custom behavior for fundamental operations on objects (property access, assignment, enumeration, function invocation, etc.):

```
const handler = {
  get(target, property) {
    return property in target ?
      target[property] :
      `Property "${property}" does not exist`;
  }
};

const user = { name: 'John' };
const proxy = new Proxy(user, handler);

console.log(proxy.name); // "John"
console.log(proxy.age);  // "Property "age" does not exist"
```

132. What are Reflect methods?

Reflect is a built-in object that provides methods for interceptable JavaScript operations, often used with Proxies. All methods that exist on Reflect correspond to the traps available to Proxy handlers.

```
// Using Reflect with Proxy
const handler = {
  get(target, prop, receiver) {
    console.log(`Getting ${prop}`);
    return Reflect.get(target, prop, receiver);
  }
};

const user = { name: 'John' };
const proxy = new Proxy(user, handler);
console.log(proxy.name); // Logs: "Getting name" then "John"
```

133. What are modules in JavaScript?

Modules are a way to organize code into separate files with their own scope, making it easier to maintain and reuse code. ES6 introduced a standardized module format with `import` and `export` statements.

```
// In utils.js
export function formatDate(date) {
  return date.toLocaleDateString();
}

export const API_URL = 'https://api.example.com';

// In main.js
import { formatDate, API_URL } from './utils.js';
```

134. What is the difference between import and require?

- **import/export**: ES6 module system, static, parsed at compile time
- **require/module.exports**: CommonJS module system, dynamic, evaluated at runtime

Main differences:

- `import` is static and must be at the top level; `require` is dynamic and can be used anywhere
- `import` can selectively load parts of a module; `require` loads the entire module
- `import` supports tree-shaking; `require` does not

135. What are dynamic imports?

Dynamic imports allow you to import modules on demand (lazily) using the `import()` function, which returns a Promise:

```
button.addEventListener('click', async () => {
  try {
    // Module loaded only when button is clicked
    const { default: Chart } = await import('./chart.js');
    const chart = new Chart();
    chart.render('#container');
  } catch (error) {
    console.error('Failed to load chart module:', error);
  }
});
```

136. What are optional chaining and nullish coalescing?

Optional chaining (`?.`) allows reading properties from deeply nested objects without explicitly checking if each reference in the chain is valid:

```
// Instead of: user && user.address && user.address.street
const street = user?.address?.street; // undefined if any part is
null/undefined
```

Nullish coalescing (`??`) provides a default value when dealing with null or undefined (but not other falsy values):

```
// Uses defaultValue only if value is null or undefined
const result = value ?? defaultValue;

// Different from:
const result = value || defaultValue; // Uses defaultValue for any falsy
value
```

137. What are BigInt numbers?

BigInt is a built-in object that provides a way to represent integers of arbitrary precision, exceeding the safe integer limit for Number (2^53 - 1):

```
const bigNumber = 9007199254740991n; // 'n' suffix for BigInt literals
const result = bigNumber + 1n;       // 9007199254740992n

// Can also create from String or Number
const bigFromString = BigInt("9007199254740993");
const bigFromNumber = BigInt(9007199254740993); // May lose precision
```

138. What are class fields and private class features?

Class fields allow instance properties to be declared directly inside the class body:

```
class User {
  // Public fields
  name = 'Anonymous';
  role = 'user';

  // Private fields (prefixed with #)
  #token = '';

  constructor(name, token) {
    if (name) this.name = name;
    this.#token = token;
  }

  // Public method
  getInfo() {
    return `${this.name} (${this.role})`;
  }

  // Private method
  #validateToken() {
    return this.#token.length > 0;
  }

  checkAccess() {
    return this.#validateToken();
  }
}
```

139. What are tagged template literals?

Tagged template literals are template literals attached to a function (tag), which processes the template literal and returns a result:

```
function highlight(strings, ...values) {
  return strings.reduce((result, string, i) => {
    return result + string + (values[i] ? `<strong>${values[i]}</strong>` :
'');
  }, '');
}

const name = 'Alice';
const age = 30;
const output = highlight`My name is ${name} and I am ${age} years old.`;
// "My name is <strong>Alice</strong> and I am <strong>30</strong> years
old."
```

140. What are the new array methods introduced in ES6+?

ES6+ introduced several new array methods:

- `Array.from()`: Creates a new Array from an array-like or iterable object
- `Array.of()`: Creates a new Array with a variable number of arguments
- `Array.prototype.find()`: Returns the first element that satisfies a condition
- `Array.prototype.findIndex()`: Returns the index of the first element that satisfies a condition
- `Array.prototype.includes()`: Determines if an array includes a value
- `Array.prototype.flat()`: Creates a new array with sub-array elements concatenated
- `Array.prototype.flatMap()`: Maps each element and flattens the result
- `Array.prototype.at()`: Returns the element at a specified index (ES2022)

Advanced Concepts

141. What is memoization in JavaScript?

Memoization is an optimization technique that stores the results of expensive function calls and returns the cached result when the same inputs occur again:

```
function memoize(fn) {
  const cache = new Map();

  return function(...args) {
    const key = JSON.stringify(args);
    if (cache.has(key)) {
      return cache.get(key);
    }

    const result = fn.apply(this, args);
    cache.set(key, result);
    return result;
  };
}

// Example: Memoized fibonacci function
const fibonacci = memoize(function(n) {
  if (n <= 1) return n;
  return fibonacci(n - 1) + fibonacci(n - 2);
});
```

142. What is currying in JavaScript?

Currying is a technique of transforming a function that takes multiple arguments into a sequence of functions that each take a single argument:

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

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

// Usage
add(1, 2, 3);           // 6
curriedAdd(1)(2)(3);    // 6

// Partial application
const add1 = curriedAdd(1);
const add1and2 = add1(2);
add1and2(3);            // 6
```

143. What is throttling and debouncing?

Throttling limits how often a function can be called in a given time period:

```
function throttle(fn, delay) {
  let lastCall = 0;

  return function(...args) {
    const now = Date.now();
    if (now - lastCall >= delay) {
      lastCall = now;
      return fn.apply(this, args);
    }
  };
}

// Usage
const throttledScroll = throttle(() => {
  console.log('Scroll event throttled');
}, 1000); // Execute at most once per second

window.addEventListener('scroll', throttledScroll);
```

Debouncing ensures a function is only called after a certain amount of time has passed since it was last called:

```
function debounce(fn, delay) {
  let timeoutId;

  return function(...args) {
    clearTimeout(timeoutId);
    timeoutId = setTimeout(() => {
      fn.apply(this, args);
    }, delay);
  };
}

// Usage
const debouncedSearch = debounce((query) => {
  console.log(`Searching for: ${query}`);
}, 500); // Wait 500ms after user stops typing

searchInput.addEventListener('input', (e) => {
  debouncedSearch(e.target.value);
});
```

144. What is the difference between throttling and debouncing?

- **Throttling**: Executes a function at most once every X milliseconds (like a rate limit)
- **Debouncing**: Executes a function only after X milliseconds have passed since it was last called (waits for a pause)

Use cases:

- **Throttle**: For continuous events where you want regular updates (scroll animations, game loop)

- **Debounce**: For events where you only care about the final state (autocomplete, resize handlers)

145. What is a pure function?

A pure function:

1. Always returns the same output given the same inputs
2. Has no side effects (doesn't modify external state)
3. Doesn't rely on external state that might change

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

// Impure function (uses external state)
let counter = 0;
function increment() {
  counter++;
  return counter;
}
```

Benefits of pure functions:

- Easier to test and debug
- Can be memoized
- Can be safely parallelized
- Promotes referential transparency

146. What is functional programming in JavaScript?

Functional programming is a programming paradigm that treats computation as the evaluation of mathematical functions and avoids changing state and mutable data. Key concepts include:

- **First-class functions**: Functions can be assigned to variables, passed as arguments, and returned from other functions
- **Pure functions**: Functions without side effects that return the same output for the same input
- **Immutability**: Once data is created, it cannot be changed
- **Higher-order functions**: Functions that operate on other functions
- **Function composition**: Building complex functions from simpler ones

```
// Example of functional programming in JavaScript
const numbers = [1, 2, 3, 4, 5];

// Using pure functions and function composition
const double = x => x * 2;
const isEven = x => x % 2 === 0;
const sum = (a, b) => a + b;

const result = numbers
```

```
  .map(double)      // [2, 4, 6, 8, 10]
  .filter(isEven)   // [2, 4, 6, 8, 10]
  .reduce(sum, 0);  // 30
```

147. What is the difference between imperative and declarative programming?

Imperative programming focuses on how to achieve something by defining step-by-step instructions:

```
// Imperative approach to sum even numbers
const numbers = [1, 2, 3, 4, 5];
let sum = 0;

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

Declarative programming focuses on what to achieve by describing the desired result:

```
// Declarative approach to sum even numbers
const numbers = [1, 2, 3, 4, 5];
const sum = numbers
  .filter(number => number % 2 === 0)
  .reduce((total, number) => total + number, 0);
```

Declarative code is often more concise, easier to understand, and less prone to bugs. It abstracts away the implementation details, making the code's intent clearer.

148. What are Web Components?

Web Components are a set of web platform APIs that allow you to create custom, reusable encapsulated HTML elements. They consist of four main technologies:

1. **Custom Elements**: JavaScript APIs to define new HTML elements
2. **Shadow DOM**: Encapsulated DOM and styling
3. **HTML Templates**: HTML fragments that aren't rendered until activated
4. **ES Modules**: JavaScript modules for loading component code

```
// Custom Element
class UserCard extends HTMLElement {
  constructor() {
    super();
    // Create Shadow DOM
    const shadow = this.attachShadow({ mode: 'open' });

    // Create elements
    const wrapper = document.createElement('div');
    wrapper.setAttribute('class', 'user-card');

    const name = document.createElement('h3');
    name.textContent = this.getAttribute('name');
    // Style with encapsulated CSS
    const style = document.createElement('style');
    style.textContent = `
      .user-card {
        border: 1px solid #ccc;
        padding: 10px;
      }
    `;

    // Append to shadow DOM
    shadow.appendChild(style);
    wrapper.appendChild(name);
    shadow.appendChild(wrapper);
  }
}

// Register the custom element
customElements.define('user-card', UserCard);
```

Usage:

```
<user-card name="John Doe"></user-card>
```

149. What is Shadow DOM?

Shadow DOM is a web standard that provides encapsulation for DOM trees and styles, allowing components to have their own isolated DOM without worrying about conflicts with the main document:

```
// Creating and using Shadow DOM
class CustomComponent extends HTMLElement {
  constructor() {
    super();

    // Create a shadow root
    const shadow = this.attachShadow({ mode: 'open' });

    // Elements created inside Shadow DOM are not affected by
    // styles from the main document, and vice versa
    const wrapper = document.createElement('div');
    wrapper.setAttribute('class', 'wrapper');

    const style = document.createElement('style');
    style.textContent = `
      /* These styles only apply within this component */
      .wrapper { background-color: #f0f0f0; }
    `;

    shadow.appendChild(style);
    shadow.appendChild(wrapper);
  }
}

customElements.define('custom-component', CustomComponent);
```

150. What are custom elements?

Custom elements are a web components API that allows developers to define their own HTML elements with custom behavior. They can be created by extending the HTMLElement class and registered using `customElements.define()`. Custom elements enable the creation of reusable, encapsulated components with their own lifecycle methods like `connectedCallback`, `disconnectedCallback`, `attributeChangedCallback`, and `adoptedCallback`.

151. What is server-side rendering vs. client-side rendering?

Server-side rendering (SSR) generates the complete HTML on the server before sending it to the client. This typically results in faster initial page loads and better SEO since search engines can easily crawl the content.

Client-side rendering (CSR) sends minimal HTML to the browser, with JavaScript handling the rendering of content in the client's browser. This approach offers better interactivity after initial load but can have slower initial render times and potential SEO challenges.

152. What is tree shaking?

Tree shaking is an optimization technique that removes unused code (dead code elimination) from the final bundle during the build process. It works by analyzing the import and export statements in ES modules to determine which code is actually being used. This significantly reduces the size of JavaScript bundles sent to the browser, improving load times and performance.

153. What is code splitting?

Code splitting is a technique that breaks down large JavaScript bundles into smaller chunks that can be loaded on demand or in parallel. Instead of loading the entire application code upfront, code splitting allows for loading only what's necessary for the current view, improving initial load performance. This is commonly implemented using dynamic imports (`import()`) or build tool configurations like Webpack's `SplitChunksPlugin`.

154. What is lazy loading?

Lazy loading is a design pattern that defers the loading of non-critical resources until they are needed. For JavaScript, this often means loading certain components, modules, or assets only when a user interacts with a feature that requires them or when they scroll to a particular section of the page. This improves initial page load performance and reduces unnecessary bandwidth usage.

155. What is the difference between compilation and transpilation?

Compilation transforms source code from one language to another, typically from a high-level language to a lower-level language (e.g., C++ to machine code).

Transpilation is a specific type of compilation that transforms source code from one language to another at the same level of abstraction (e.g., TypeScript to JavaScript, or ES6+ to ES5). Transpilation is commonly used in JavaScript development to convert modern code into versions compatible with older browsers.

156. What is the purpose of Babel?

Babel is a JavaScript transpiler that converts modern JavaScript code (ES6+) into backward-compatible versions that can run in older browsers. It allows developers to use the latest JavaScript features without worrying about browser compatibility. Babel is highly configurable through plugins and presets, enabling selective transformation of specific language features.

157. What is webpack and how does it work?

Webpack is a module bundler for JavaScript applications. It builds a dependency graph by analyzing the imports and exports in your code, then combines all the necessary files into optimized bundles for production.

Webpack works through:

1. **Entry**: Defining the starting point(s) of the application
2. **Output**: Specifying where to emit the bundles
3. **Loaders**: Transforming non-JavaScript files into modules
4. **Plugins**: Performing a wider range of tasks like optimization, asset management, and environment variable injection
5. **Mode**: Applying built-in optimizations based on development or production environment

158. What is the purpose of polyfills?

Polyfills are code snippets that provide modern functionality on older browsers that don't support certain features natively. They "fill in" the gaps in browser implementations by adding the missing features, allowing developers to use new JavaScript APIs and syntax while maintaining compatibility with older browsers. For example, polyfills can add support for `Promise`, `fetch`, or `Array.prototype.includes` in browsers that don't have these features built-in.

159. What is TypeScript and why use it?

TypeScript is a strongly typed superset of JavaScript that compiles to plain JavaScript. It adds optional static typing, interfaces, advanced type checking, and other features to JavaScript.

Reasons to use TypeScript:

- **Type safety**: Catches type-related errors during development

- **Better IDE support**: Enhanced autocomplete, navigation, and refactoring
- **Improved maintainability**: Self-documenting code through type annotations
- **Easier refactoring**: Types make large-scale code changes safer
- **Enhanced team collaboration**: Clear interfaces and contracts between components

160. What are decorators in JavaScript?

Decorators are a design pattern and experimental JavaScript feature (part of TC39 proposal) that allow for modifying or enhancing classes and class members declaratively. They are functions that return a function and can be applied using the `@decorator` syntax above classes, methods, properties, or parameters.

Decorators are commonly used in frameworks like Angular and libraries like MobX to implement features such as dependency injection, observables, memoization, and access control. While native JavaScript decorators are still in the proposal stage, TypeScript has supported them for some time.

Performance and Best Practices

161. How do you optimize JavaScript performance?

Key strategies for optimizing JavaScript performance include:

- Minimize DOM manipulations by batching updates or using document fragments
- Reduce render-blocking JavaScript with async/defer attributes
- Implement code splitting and lazy loading
- Use efficient data structures and algorithms
- Optimize loops and recursion
- Avoid memory leaks by properly managing event listeners and closures
- Leverage browser caching and CDNs
- Use Web Workers for CPU-intensive tasks
- Implement virtualization for long lists
- Apply proper debouncing and throttling for event handlers
- Optimize animations with requestAnimationFrame
- Use performance profiling tools to identify bottlenecks

162. What causes memory leaks in JavaScript?

Common causes of memory leaks in JavaScript include:

- **Uncleared event listeners**: Adding event listeners without removing them when no longer needed
- **Closures capturing large objects**: Functions that maintain references to large objects
- **Detached DOM elements**: DOM elements removed from the document but still referenced in JavaScript
- **Global variables**: Accumulating data in global scope
- **Circular references**: Objects referencing each other in a way that prevents garbage collection
- **Timers and intervals**: setInterval or setTimeout callbacks that aren't cleared

- **Cache implementations**: Storing large amounts of data without proper cleanup strategies

163. How do you measure JavaScript performance?

Performance can be measured using various tools and techniques:

- **Browser DevTools**: Performance and Memory panels for profiling
- **Performance API**: Using `performance.now()` for precise timing measurements
- **Lighthouse**: Automated auditing for web performance
- **WebPageTest**: External testing of page load metrics
- **Chrome User Experience Report**: Real-world performance data
- **Custom performance markers**: Using `performance.mark()` and `performance.measure()`
- **Core Web Vitals**: Metrics like LCP, FID, and CLS
- **Memory profilers**: To identify memory consumption patterns and leaks
- **Bundle analyzers**: To examine and optimize bundle sizes

164. What is the difference between defer and async script loading?

async: Scripts load asynchronously while HTML parsing continues. They execute as soon as they're downloaded, potentially interrupting HTML parsing. The order of execution is not guaranteed to match the order in the document.

defer: Scripts also load asynchronously without blocking HTML parsing, but they execute in order only after HTML parsing is complete and before the DOMContentLoaded event. This preserves execution order while still providing performance benefits.

Regular `<script>` tags without these attributes block HTML parsing until the script is downloaded and executed.

165. How do you debug JavaScript code?

JavaScript debugging techniques include:

- Using `console.log()`, `console.error()`, and other console methods
- Setting breakpoints in browser DevTools or with the `debugger` statement
- Using the Sources panel in DevTools to step through code execution
- Utilizing watch expressions to monitor variable values
- Using conditional breakpoints for specific scenarios
- Examining call stacks to understand execution flow
- Using try/catch blocks to handle and log errors
- Leveraging source maps for debugging transpiled code
- Implementing error tracking services like Sentry
- Using browser extensions like React/Redux DevTools for framework-specific debugging

166. What are common JavaScript design patterns?

Common JavaScript design patterns include:

- **Module Pattern**: Encapsulating code in private/public scopes
- **Singleton Pattern**: Ensuring a class has only one instance
- **Factory Pattern**: Creating objects without specifying their concrete class
- **Observer Pattern**: Creating subscription-based notification systems
- **Decorator Pattern**: Adding new functionality to existing objects dynamically
- **Strategy Pattern**: Defining a family of interchangeable algorithms
- **Command Pattern**: Encapsulating a request as an object
- **Proxy Pattern**: Controlling access to an object
- **Facade Pattern**: Providing a simplified interface to complex subsystems
- **Mediator Pattern**: Defining an object that encapsulates how objects interact

167. What is the module pattern?

The module pattern is a design pattern that provides a way to encapsulate related code into a single unit with private and public members. It creates a closure around the module's variables and functions, providing privacy and preventing pollution of the global namespace.

Implementation typically involves an IIFE (Immediately Invoked Function Expression) that returns an object containing public methods and properties, while keeping private functionality hidden inside the closure.

```
const myModule = (function() {
  // Private variables and functions
  let privateVar = 'I am private';

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

  // Public API
  return {
    publicVar: 'I am public',
    publicMethod: function() {
      privateMethod();
    }
  };
})();
```

168. What is the singleton pattern?

The singleton pattern ensures a class has only one instance and provides a global point of access to it. It's useful when exactly one object is needed to coordinate actions across the system, such as a configuration manager or connection pool.

```
const Singleton = (function() {
  let instance;

  function createInstance() {
    const object = new Object("I am the singleton instance");
    return object;
  }
  return {
    getInstance: function() {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
  };
})();
```

169. What is the observer pattern?

The observer pattern defines a one-to-many dependency between objects where when one object (the subject) changes state, all its dependents (observers) are notified and updated automatically. This is essential for implementing distributed event handling systems and is the foundation for many aspects of reactive programming.

```
class Subject {
  constructor() {
    this.observers = [];
  }

  subscribe(observer) {
    this.observers.push(observer);
  }

  unsubscribe(observer) {
    this.observers = this.observers.filter(obs => obs !== observer);
  }

  notify(data) {
    this.observers.forEach(observer => observer.update(data));
  }
}

class Observer {
  update(data) {
    console.log(`Updated with data: ${data}`);
  }
}
```

170. What is the factory pattern?

The factory pattern is a creational pattern that provides an interface for creating objects without specifying their concrete classes. It encapsulates object creation logic, allowing for flexibility in what objects are created based on different parameters or conditions.

```
class UserFactory {
  createUser(type) {
    switch(type) {
      case 'admin':
        return new AdminUser();
      case 'regular':
        return new RegularUser();
      case 'guest':
        return new GuestUser();
      default:
        throw new Error('Invalid user type');
    }
  }
}
```

171. How do you handle error handling in JavaScript?

Effective error handling in JavaScript involves:

- Using try/catch blocks for synchronous code
- Using .catch() for promises or try/catch with async/await
- Implementing global error handlers with window.onerror or unhandledrejection
- Creating custom error classes by extending the Error class
- Providing meaningful error messages and context
- Logging errors for debugging while showing user-friendly messages
- Implementing defensive programming techniques
- Using type checking and validation
- Failing fast with early returns and guard clauses
- Implementing error boundaries in UI frameworks

172. What are linters and why are they useful?

Linters are static code analysis tools that flag programming errors, bugs, stylistic errors, and suspicious constructs. Popular JavaScript linters include ESLint, JSLint, and JSHint.

Linters are useful because they:

- Enforce consistent coding conventions and style
- Catch potential errors and bugs early in development
- Identify code quality issues and anti-patterns
- Help maintain cleaner, more readable code
- Reduce technical debt
- Provide automatic fixes for common issues
- Enforce team coding standards
- Prevent problematic patterns specific to JavaScript
- Integrate with code editors for real-time feedback
- Can be customized to project-specific requirements

173. What is the importance of code formatting?

Code formatting is important because it:

- Improves readability and comprehension
- Makes codebases more maintainable over time
- Reduces cognitive load for developers
- Facilitates collaboration in team environments
- Minimizes merge conflicts in version control
- Creates consistency across a project
- Makes code reviews more efficient
- Helps identify logical errors that become apparent with proper indentation

- Establishes professional coding standards
- Allows developers to focus on solving problems rather than syntax

Tools like Prettier automate code formatting to enforce consistent style rules.

174. How do you write testable JavaScript code?

To write testable JavaScript code:

- Follow SOLID principles (especially Single Responsibility)
- Use dependency injection to isolate units of code
- Minimize side effects and make functions pure where possible
- Separate business logic from UI/DOM manipulation
- Use modular design with clear interfaces
- Avoid global state and prefer passing parameters
- Write small, focused functions with clear responsibilities
- Make outputs depend only on inputs and avoid hidden dependencies
- Use interfaces and abstractions to mock dependencies
- Implement inversion of control patterns
- Keep concerns separated (data access, business logic, presentation)

175. What are unit tests, integration tests, and E2E tests?

Unit tests focus on testing individual components (functions, methods, classes) in isolation. Dependencies are typically mocked or stubbed. Tools include Jest, Mocha, and Jasmine.

Integration tests verify that different modules or services work together correctly. They test the interactions between components or systems. Tools include Supertest for API testing or Jest with partial mocks.

End-to-End (E2E) tests validate entire workflows from start to finish from a user's perspective. They test the application in an environment that mimics real-world usage. Tools include Cypress, Playwright, and Selenium.

Each type of test serves different purposes in a comprehensive testing strategy, forming a testing pyramid with many unit tests at the base, fewer integration tests in the middle, and even fewer E2E tests at the top.

176. What are JavaScript best practices for security?

JavaScript security best practices include:

- Validate and sanitize all user inputs
- Use Content Security Policy (CSP) headers
- Implement proper authentication and authorization
- Avoid using eval() and Function constructors
- Sanitize HTML content before inserting into DOM
- Use HTTPS for all connections
- Implement proper CORS policies

- Be cautious with third-party libraries and dependencies
- Apply the principle of least privilege
- Use secure cookies with httpOnly, secure, and sameSite flags
- Protect against CSRF attacks with tokens
- Implement rate limiting for APIs
- Keep dependencies updated for security patches
- Use secure password hashing algorithms
- Avoid storing sensitive data in localStorage or sessionStorage

177. How do you prevent XSS attacks?

To prevent Cross-Site Scripting (XSS) attacks:

- Escape or sanitize user-generated content before rendering to HTML
- Use libraries like DOMPurify to sanitize HTML
- Implement Content Security Policy (CSP) headers
- Use templating libraries that automatically escape content
- Avoid using dangerous functions like innerHTML, document.write, and eval()
- Prefer textContent over innerHTML when not needing HTML rendering
- Use HTTPOnly cookies to prevent JavaScript access to sensitive cookies
- Implement input validation on both client and server
- Use frameworks that automatically escape output (React, Vue, Angular)
- Apply the principle of least privilege to JavaScript execution
- Keep all libraries and frameworks updated

178. What is CORS and how does it work?

CORS (Cross-Origin Resource Sharing) is a security mechanism that allows web servers to specify which origins can access their resources. It works through HTTP headers that define which cross-origin requests are permitted.

Key components of CORS:

- **Origin header**: Sent by the browser to indicate the requesting origin
- **Access-Control-Allow-Origin**: Server response header specifying allowed origins
- **Preflight requests**: OPTIONS requests sent before actual requests for non-simple requests
- **Access-Control-Allow-Methods**: Specifies allowed HTTP methods
- **Access-Control-Allow-Headers**: Indicates which headers can be included in requests
- **Access-Control-Allow-Credentials**: Controls whether credentials can be included in requests
- **Access-Control-Max-Age**: Specifies how long preflight results can be cached

CORS prevents malicious websites from making unauthorized requests to APIs or services on behalf of users.

179. What is the Same-Origin Policy?

The Same-Origin Policy is a critical security mechanism implemented by web browsers that restricts how documents or scripts from one origin can interact with resources from another origin. An origin is defined by the combination of protocol (http/https), domain, and port.

Under this policy:

- JavaScript can freely access resources from the same origin
- Scripts can make cross-origin requests, but cannot read the responses without proper CORS headers
- Scripts cannot directly access DOM or JavaScript objects from different origins
- Cookies, localStorage, and IndexedDB are segregated by origin

This policy is fundamental to web security as it prevents malicious websites from reading sensitive data from other websites the user is logged into.

180. How do you prevent CSRF attacks?

To prevent Cross-Site Request Forgery (CSRF) attacks:

- Implement anti-CSRF tokens (unique tokens embedded in forms that are validated on submission)
- Use the SameSite cookie attribute (Lax or Strict) to limit cookie transmission
- Verify the Origin and Referer headers for sensitive actions
- Require re-authentication for sensitive operations
- Implement proper CORS policies to limit cross-origin requests
- Use custom request headers that simple forms cannot create
- Add CAPTCHA for sensitive operations
- Use POST requests instead of GET for state-changing operations
- Apply proper session management
- Implement request rate limiting

Frameworks and Libraries

181. What is the difference between a library and a framework?

Library: A collection of pre-written code that provides specific functionality that developers can call when needed. Libraries are integrated into existing projects and developers control when and where to use them. Examples include jQuery, Lodash, and Moment.js.

Framework: A more comprehensive structure that provides a foundation for developing applications. Frameworks often implement the "Inversion of Control" principle where the framework calls the developer's code rather than the other way around. Frameworks typically dictate application architecture and flow. Examples include React, Angular, and Vue.js.

The key difference lies in control: with libraries, you control the flow of your application and call the library when needed; with frameworks, the framework controls the flow and calls your code when appropriate.

182. What is React and what are its key features?

React is a JavaScript library for building user interfaces, particularly single-page applications. Developed by Facebook, it focuses on the view layer of the MVC pattern.

Key features of React:

- **Virtual DOM**: Efficiently updates the actual DOM by comparing with a lightweight virtual representation
- **Component-Based Architecture**: Encapsulates UI pieces into reusable, composable components
- **JSX**: Syntax extension allowing HTML-like code in JavaScript
- **Unidirectional Data Flow**: Data flows down from parent to child components
- **React Hooks**: Functions that let you use state and lifecycle features in functional components
- **React Router**: Handles navigation in single-page applications
- **React Context**: Manages global state without prop drilling
- **Server-Side Rendering**: Supports rendering on the server for improved performance and SEO
- **Large Ecosystem**: Rich library of tools, extensions, and community support
- **Developer Tools**: Comprehensive debugging and development tools

183. What is Vue.js and what are its key features?

Vue.js is a progressive JavaScript framework for building user interfaces and single-page applications. Created by Evan You, it's designed to be incrementally adoptable.

Key features of Vue.js:

- **Reactive Data Binding**: Automatically updates the DOM when data changes
- **Component System**: Allows building encapsulated, reusable components
- **Template Syntax**: HTML-based template syntax that's easy to understand
- **Vue Router**: Official router for building single-page applications
- **Vuex**: State management pattern and library
- **Directives**: Special attributes with the `v-` prefix for DOM manipulations
- **Two-way Binding**: Makes form input and application state synchronization easy
- **Transitions and Animations**: Built-in transition system for CSS animations
- **Computed Properties**: Derived values that are cached based on dependencies
- **Watchers**: React to data changes with custom logic
- **Vue CLI**: Standard tooling for Vue.js development
- **Single-File Components**: .vue files containing template, script, and style

184. What is Angular and what are its key features?

Angular is a comprehensive TypeScript-based open-source web application framework led by the Angular Team at Google. It's a complete rewrite of AngularJS.

Key features of Angular:

- **TypeScript Integration**: Built with and encourages the use of TypeScript
- **Component-Based Architecture**: Organizes code into reusable components
- **Dependency Injection**: Built-in powerful DI system
- **Directives**: Extend HTML with custom attributes and elements
- **Templates**: HTML-based templates with Angular-specific syntax

- **Pipes**: Transform data for display in templates
- **Forms**: Robust form handling with template-driven and reactive approaches
- **HttpClient**: Communication with backend services via HTTP
- **Router**: Advanced client-side navigation and routing
- **Observables**: Uses RxJS for handling asynchronous events
- **Angular CLI**: Command-line tools for development workflow
- **Testing Utilities**: Comprehensive testing framework
- **AOT Compilation**: Ahead-of-Time compilation for faster rendering
- **Angular Material**: UI component library implementing Material Design

185. What is Node.js and how does it differ from browser JavaScript?

Node.js is an open-source, cross-platform JavaScript runtime environment that executes JavaScript code outside a web browser. It uses Chrome's V8 JavaScript engine.

Key differences from browser JavaScript:

- **Environment**: Node.js runs on the server-side or as a standalone application, while browser JavaScript runs in the browser
- **DOM Access**: Node.js lacks DOM and window objects since it doesn't run in a browser
- **Global Object**: Uses `global` instead of `window` as the global object
- **File System Access**: Node.js can access the file system, while browser JavaScript has limited file access due to security restrictions
- **Modules**: Node.js uses CommonJS module system (`require`), though ES modules are now supported
- **Package Management**: Uses npm or yarn for package management
- **Threading Model**: Uses a single-threaded event loop with non-blocking I/O
- **Built-in APIs**: Provides APIs for HTTP, file system, streams, and other server-side functionalities
- **Security Model**: Different security considerations (browser focus on sandbox, Node focuses on file system security)
- **Buffer**: Node.js has Buffer class for handling binary data, absent in browsers

186. What is Express.js?

Express.js is a minimal and flexible Node.js web application framework that provides a robust set of features for web and mobile applications. It's the most popular web framework for Node.js and serves as the foundation for many other Node.js web frameworks.

Key features of Express.js:

- **Middleware**: Modular middleware functions to process requests
- **Routing**: Simple route handling for different HTTP methods and URLs
- **Template Engines**: Integration with various template engines like Pug, EJS, and Handlebars
- **Static File Serving**: Integrated middleware for serving static files
- **HTTP Utility Methods**: Methods for handling HTTP requests and responses
- **REST API Support**: Simplified creation of RESTful APIs
- **Error Handling**: Built-in error handling mechanisms

- **Extensibility**: Easy to extend with plugins and middleware
- **Performance**: Lightweight and performant due to minimal abstraction
- **HTTP Helpers**: Convenience methods for HTTP headers, cookies, etc.

187. What is Redux and when should you use it?

Redux is a predictable state container for JavaScript applications, commonly used with React but compatible with any UI library. It provides a central store that holds the entire application state and enforces rules ensuring the state can only be updated in a predictable fashion.

When to use Redux:

- When managing complex application state that is shared across many components
- When the state needs to be updated by many different components
- When you need to implement features like undo/redo, state persistence, or logging
- For large applications where state management becomes complex
- When you need a predictable state management pattern
- When you want to separate state management from UI components
- When multiple actions need to update the same state

Redux may not be necessary for simpler applications where component state or Context API would suffice.

188. What is GraphQL?

GraphQL is a query language for APIs and a runtime for executing those queries by using a type system defined for the data. It was developed by Facebook in 2012 and released as an open-source project in 2015.

Key features of GraphQL:

- **Declarative Data Fetching**: Clients specify exactly what data they need
- **Single Endpoint**: All requests go through a single endpoint
- **Strongly Typed**: Schema defines available data types and operations
- **No Over-fetching or Under-fetching**: Only requested data is delivered
- **Introspection**: API can be queried for its own schema
- **Real-time Updates**: Subscriptions allow for real-time data updates
- **Version-less**: Evolve API without versioning
- **Query Batching**: Multiple queries can be sent in a single request
- **Directives**: Annotations that modify query execution
- **Client-specific Responses**: Different clients can request different data

189. What is the difference between REST and GraphQL?

Key differences between REST and GraphQL:

Endpoints:

- REST: Multiple endpoints for different resources

- GraphQL: Single endpoint for all operations

Data Fetching:

- REST: Fixed data structures returned by each endpoint
- GraphQL: Client specifies exactly what data it needs

Over-fetching/Under-fetching:

- REST: Common problem where endpoints return too much or too little data
- GraphQL: Client only gets requested data

Versioning:

- REST: Often requires versioning (e.g., /api/v1/, /api/v2/)
- GraphQL: Evolves continuously by adding fields without breaking existing queries

Caching:

- REST: HTTP caching mechanisms are well-established
- GraphQL: Requires custom caching implementation

State Modification:

- REST: Uses HTTP methods (GET, POST, PUT, DELETE)
- GraphQL: Uses mutations for all modifications

Learning Curve:

- REST: Familiar and well-established patterns
- GraphQL: Additional concepts to learn (schema, resolvers, etc.)

Network Performance:

- REST: Multiple round trips for related resources
- GraphQL: Can get all needed data in a single request

190. What are Web Sockets?

WebSockets is a communication protocol that provides full-duplex communication channels over a single TCP connection. Unlike HTTP, which is request-response based, WebSockets allow for persistent connections where both the server and client can send data at any time.

Key aspects of WebSockets:

- **Bidirectional**: Both client and server can send messages independently
- **Persistent Connection**: Connection remains open for the session duration
- **Low Latency**: Minimal overhead after initial handshake
- **Real-time Communication**: Ideal for applications requiring real-time updates
- **Protocol**: Uses `ws://` or `wss://` (secure) protocol

- **Less Overhead**: Reduced header information compared to HTTP
- **Browser Support**: Widely supported in modern browsers
- **Standardized**: Part of the HTML5 standard

Common use cases include chat applications, live feeds, collaborative editing, gaming, and real-time dashboards.

191. What is the difference between jQuery and modern JavaScript?

jQuery was created in 2006 to simplify DOM manipulation and provide cross-browser compatibility when browsers had significant inconsistencies.

Modern JavaScript refers to more recent versions of JavaScript (ES6+) and current development practices.

Key differences:

DOM Manipulation:

- jQuery: `$('#element').hide()`
- Modern JS: `document.getElementById('element').style.display = 'none'`

Browser Compatibility:

- jQuery: Built to handle cross-browser inconsistencies
- Modern JS: Modern browsers have more consistent implementations

Ajax:

- jQuery: `$.ajax()`
- Modern JS: Fetch API or Axios

Size:

- jQuery: Additional library to download
- Modern JS: Built into browsers

Animation:

- jQuery: `.animate()` method
- Modern JS: CSS animations or Web Animation API

Event Handling:

- jQuery: `.on()` method with simplified syntax
- Modern JS: `addEventListener()` with more options

Syntax:

- jQuery: Chainable methods on jQuery objects
- Modern JS: Native methods, often with different syntax

Architecture:

- jQuery: Primarily focused on DOM manipulation and AJAX
- Modern JS: Component-based architecture using frameworks like React/Vue/Angular

192. What are JavaScript testing libraries like Jest and Mocha?

Jest is a comprehensive JavaScript testing framework developed by Facebook. It's designed to ensure correctness of JavaScript code with minimal configuration.

Key features of Jest:

- Zero configuration setup
- Built-in assertion library
- Snapshot testing
- Mocking capabilities
- Parallel test execution
- Code coverage reports
- Interactive watch mode
- Isolated test environment with jsdom

Mocha is a flexible JavaScript test framework running on Node.js and in the browser.

Key features of Mocha:

- Supports various assertion libraries (Chai, Should.js)
- Supports various mocking libraries (Sinon)
- Support for asynchronous testing
- Customizable reporting
- Browser support
- Flexible plugin architecture
- Test coverage via Istanbul

Other popular testing libraries include QUnit, Jasmine, Karma, and Cypress for end-to-end testing.

193. What is the purpose of bundlers like webpack and Rollup?

Bundlers are tools that process JavaScript applications, resolving dependencies and merging (or "bundling") many files into optimized bundles for deployment.

Webpack is a module bundler primarily focused on JavaScript but capable of transforming all project assets. Its purposes include:

- Bundling modules with dependencies into static assets
- Code splitting and lazy loading
- Development server with hot module replacement
- Asset transformation via loaders (CSS, images, etc.)
- Plugin system for extended functionality
- Optimization for production (minification, tree shaking)
- Source maps for debugging

Rollup is a module bundler specializing in ES modules. Its purposes include:

- Efficient bundling with "tree shaking" to eliminate dead code
- Creating smaller bundles than traditional bundlers
- Generating various output formats (ESM, CJS, UMD, etc.)
- Code splitting capabilities
- Plugin ecosystem for transformations
- Optimized for libraries (smaller builds)

194. What are PWAs (Progressive Web Applications)?

Progressive Web Applications are web applications that use modern web capabilities to provide an app-like experience to users. They combine the best of web and native applications.

Key characteristics of PWAs:

- **Progressive**: Works for every user, regardless of browser choice
- **Responsive**: Fits any form factor (desktop, mobile, tablet)
- **Connectivity Independent**: Works offline or with poor network conditions
- **App-like**: Feels like a native app with app-style interactions
- **Fresh**: Always up-to-date due to the service worker update process
- **Safe**: Served via HTTPS to prevent snooping
- **Discoverable**: Identifiable as "applications" via W3C manifests
- **Re-engageable**: Makes re-engagement easy through push notifications
- **Installable**: Allows users to add apps to their home screen
- **Linkable**: Easily shareable via URLs, doesn't require installation

Key technologies that enable PWAs include Service Workers, Web App Manifests, and Push Notifications.

195. What is server-side JavaScript?

Server-side JavaScript refers to using JavaScript on the server to handle requests, interact with databases, manage file systems, and perform other server operations traditionally handled by languages like PHP, Python, or Java.

Key aspects of server-side JavaScript:

- **Node.js**: The most popular runtime environment for server-side JavaScript
- **Event-driven architecture**: Non-blocking I/O model for handling concurrent connections
- **NPM ecosystem**: Vast library of reusable packages

- **Common uses**: RESTful APIs, real-time applications, microservices
- **Frameworks**: Express.js, Nest.js, Koa, Fastify, Hapi
- **Database integration**: Native drivers for MongoDB, PostgreSQL, MySQL, etc.

196. What is the MEAN/MERN/MEVN stack?

These are popular full-stack JavaScript technology stacks:

MEAN Stack:

- **M**ongoDB: NoSQL database
- **E**xpress.js: Web application framework
- **A**ngular: Front-end framework
- **N**ode.js: JavaScript runtime environment

MERN Stack:

- **M**ongoDB: NoSQL database
- **E**xpress.js: Web application framework
- **R**eact: Front-end library
- **N**ode.js: JavaScript runtime environment

MEVN Stack:

- **M**ongoDB: NoSQL database
- **E**xpress.js: Web application framework
- **V**ue.js: Front-end framework
- **N**ode.js: JavaScript runtime environment

These stacks allow developers to use JavaScript throughout the entire application, from database queries to server-side logic to client-side rendering. They're popular for building modern web applications due to their flexibility, scalability, and the advantage of using a single language across the entire stack.

197. What is JAMstack?

JAMstack is a modern web development architecture based on client-side JavaScript, reusable APIs, and prebuilt Markup. The term was coined by Mathias Biilmann, CEO of Netlify.

Key components of JAMstack:

- **J**avaScript: Dynamic programming on the client side
- **A**PIs: Server-side processes abstracted into reusable APIs
- **M**arkup: Pre-built markup, typically using a Static Site Generator

Key benefits of JAMstack:

- **Better performance**: Pre-built files served over CDN

- **Higher security**: Reduced attack surface with no server or database
- **Cheaper, easier scaling**: Static hosting is simple and inexpensive
- **Better developer experience**: Decoupled architecture allows specialized focus
- **Improved reliability**: Fewer moving parts and points of failure

Common tools in the JAMstack ecosystem include static site generators (Gatsby, Next.js, Nuxt.js, Hugo), headless CMS systems (Contentful, Sanity, Strapi), and deployment platforms (Netlify, Vercel).

198. What is the role of TypeScript in modern JavaScript development?

TypeScript plays several crucial roles in modern JavaScript development:

- **Type Safety**: Provides static type checking during development, catching errors before runtime
- **Code Organization**: Interfaces, enums, and generics help structure complex codebases
- **Developer Productivity**: Enhanced tooling support with better autocompletion and navigation
- **Documentation**: Types serve as living documentation about code expectations
- **Refactoring Support**: Makes large-scale code changes safer and more predictable
- **Backwards Compatibility**: Any valid JavaScript is valid TypeScript, allowing gradual adoption
- **Future Features**: Implements proposed JavaScript features before browsers support them
- **Enterprise Adoption**: Facilitates adoption of JavaScript in larger organizations
- **Framework Integration**: Used as the foundation for Angular and heavily supported in React and Vue
- **API Development**: Provides clear contracts between client and server components

TypeScript has become increasingly important as JavaScript applications grow in size and complexity, helping manage that complexity through its type system and tooling.

199. What are Static Site Generators?

Static Site Generators (SSGs) are tools that generate static HTML, CSS, and JavaScript files at build time rather than dynamically on the server at request time. They combine content (often in Markdown format) with templates to produce complete, pre-rendered websites.

Popular JavaScript-based SSGs include:

- **Gatsby**: React-based with GraphQL data layer
- **Next.js**: React framework with static generation and server-side rendering
- **Nuxt.js**: Vue.js equivalent to Next.js
- **Eleventy (11ty)**: Simpler alternative focused on templates
- **Astro**: Multi-framework supporting React, Vue, Svelte components
- **Gridsome**: Vue-based with GraphQL

Key benefits of Static Site Generators:

- **Performance**: Fast load times as pages are pre-rendered
- **Security**: No server-side logic to exploit
- **Scalability**: Easy and inexpensive to host and distribute via CDNs
- **Developer Experience**: Leverage modern tools while outputting optimized sites
- **SEO Friendly**: Pre-rendered content is easily indexable by search engines
- **Content Management**: Can be paired with headless CMS systems
- **Version Control**: Static files can be version-controlled

200. What are the current trends in JavaScript development?

Current trends in JavaScript development include:

Technical trends:

- **TypeScript Adoption**: Increasing use of TypeScript for type safety
- **Framework Evolution**: Continued improvement of React, Vue, Angular with focus on performance
- **Build Tools**: Evolution of build tools (Vite, esbuild, Turbopack) with faster build times
- **Server Components**: Server-side rendering and hydration improvements (React Server Components)
- **Edge Computing**: JavaScript running on edge networks (Cloudflare Workers, Deno Deploy)
- **WebAssembly Integration**: Using WASM with JavaScript for performance-critical code
- **Micro-frontends**: Breaking applications into independently deployable modules
- **State Management**: Simplification with React Context, Redux Toolkit, Zustand, Jotai
- **Meta-frameworks**: Growth of Next.js, Nuxt, Remix, and other opinionated frameworks
- **Islands Architecture**: Partial hydration strategies for more efficient client-side JavaScript

Architectural trends:

- **JAMstack/Static-first**: Pre-rendering with dynamic capabilities
- **Headless CMS**: Separation of content management from presentation
- **API-first Development**: Focus on building and consuming well-designed APIs
- **Serverless**: Growth of serverless functions for backend logic
- **Component-driven Design**: Building UIs from composable components
- **Design Systems**: Standardized component libraries for consistent interfaces

Developer experience trends:

- **AI-assisted Development**: Tools like GitHub Copilot for code suggestions
- **Zero-config Tools**: Simplified tooling with smart defaults

- **Containerization**: Standardized development environments with Docker
- **Monorepos**: Managing multiple packages in a single repository
- **Testing Emphasis**: Growing focus on automated testing at all levels
- **Web Standards**: Greater alignment with browser capabilities