JAVASCRIPT INTERVIEW QUESTIONS

1.1 Variables and Data Types

- **Q:** What are the different data types in JavaScript?
    - **A:** JavaScript has six primitive data types: string, number, boolean, undefined, null, and symbol. Additionally, object is a non-primitive data type.
- **Q:** What is the difference between let, var, and const?
    - **A:** var is function-scoped and can be redeclared and updated. let is block-scoped, cannot be redeclared but can be updated. const is block-scoped, cannot be redeclared or updated.
- **Q:** What is hoisting in JavaScript?
    - **A:** Hoisting is JavaScript's default behavior of moving declarations to the top of the current scope. Both var variables and function declarations are hoisted, but let and const variables are not initialized until they are encountered in the code.

1.2 Functions

- **Q:** What is a first-class function?
    - **A:** JavaScript treats functions as first-class citizens, meaning they can be assigned to variables, passed as arguments to other functions, and returned from functions.
- **Q:** What is a higher-order function?
    - **A:** A higher-order function is a function that takes another function as an argument or returns a function as a result.

- **Q:** Explain the difference between function declaration and function expression.
    - **A:** Function declarations are hoisted, meaning they can be called before they are defined in the code. Function expressions are not hoisted and cannot be called before they are defined.

2.1 Scope and Closures

- **Q:** What is the difference between global scope and local scope?
    - **A:** Global scope refers to variables that are accessible anywhere in the code, whereas local scope refers to variables that are accessible only within a specific function or block.
- **Q:** What is a closure?
    - **A:** A closure is a function that remembers and has access to variables from the lexical scope in which it was created, even after that scope has exited.
- **Q:** How can you prevent a variable from being modified in a closure?
    - **A:** You can use const to declare the variable inside the closure, making it read-only within that scope.

2.2 Asynchronous JavaScript

- **Q:** What is the difference between setTimeout and setInterval?
    - **A:** setTimeout executes a function once after a specified delay, while setInterval executes a function repeatedly with a fixed time delay between each call.

- **Q:** Explain the concept of promises.
    - **A:** A promise is an object representing the eventual completion or failure of an asynchronous operation. It can be in one of three states: pending, fulfilled, or rejected.
- **Q:** What is the async/await syntax?
    - **A:** async/await is syntactic sugar built on top of promises, allowing you to write asynchronous code that looks synchronous. await pauses the execution of the function until the promise is resolved or rejected.

3.1 Prototypes and Inheritance

- **Q:** What is prototypal inheritance?
    - **A:** Prototypal inheritance is a feature in JavaScript where objects inherit properties and methods from other objects. This is achieved through the prototype chain.
- **Q:** How can you implement inheritance in JavaScript?
    - **A:** Inheritance in JavaScript can be implemented using the prototype property, Object.create() method, or ES6 classes with the extends keyword.
- **Q:** What is the difference between classical inheritance and prototypal inheritance?
    - **A:** Classical inheritance (used in languages like Java) is based on classes and involves creating instances of these classes. Prototypal inheritance is based on objects inheriting directly from other objects.

3.2 Advanced Functions and Callbacks

- **Q:** What are immediately invoked function expressions (IIFE)?
    - **A:** IIFE is a function that is executed immediately after its creation. It is used to create a private scope to avoid polluting the global namespace.
- **Q:** Explain the call, apply, and bind methods.
    - **A:**
        - call: Invokes a function with a given this value and arguments passed individually.
        - apply: Similar to call, but arguments are passed as an array.
        - bind: Returns a new function, permanently binding this to the provided context.
- **Q:** What is currying in JavaScript?
    - **A:** Currying is the process of transforming a function that takes multiple arguments into a series of functions that each take a single argument.

3.3 Event Loop and Concurrency

- **Q:** What is the event loop in JavaScript?
    - **A:** The event loop is a mechanism that handles asynchronous operations by continuously checking the call stack and task queue, executing code, collecting and processing events, and executing queued sub-tasks.

- **Q:** Explain the difference between microtasks and macrotasks.
    - **A:** Microtasks (e.g., promises) are executed immediately after the current task completes and before the event loop checks for the next macrotask (e.g., setTimeout). Macrotasks are executed after all microtasks are completed.
- **Q:** What is a Web Worker, and how does it help in concurrency?
    - **A:** A Web Worker is a JavaScript script that runs in the background, independent of the main execution thread, allowing you to perform computationally expensive tasks without blocking the UI.

4.1 Advanced Object Manipulation

- **Q:** What is the difference between deep copy and shallow copy in JavaScript?
    - **A:** A shallow copy duplicates only the top-level properties of an object, while a deep copy duplicates all nested objects and arrays as well.
- **Q:** How can you achieve a deep copy of an object in JavaScript?
    - **A:** You can achieve a deep copy using methods like JSON.parse(JSON.stringify(obj)), lodash's cloneDeep, or a recursive function to copy each nested object.

4.2 Memory Management and Performance

- **Q:** What is garbage collection in JavaScript?
    - **A:** Garbage collection is an automatic process by which the JavaScript engine frees up memory by removing objects that are no longer reachable or needed in the application.
- **Q:** Explain the concept of memory leaks in JavaScript.

    - **A:** Memory leaks occur when memory that is no longer needed is not released, leading to increased memory usage over time. Common causes include forgotten timers, closures, and global variables.

4.3 Design Patterns

- **Q:** What is the Module Pattern in JavaScript?
    - **A:** The Module Pattern is a design pattern used to create private and public methods and variables, encapsulating code within a single closure to avoid polluting the global namespace.
- **Q:** What is the Observer Pattern in JavaScript?
    - **A:** The Observer Pattern involves objects known as observers that are notified of changes to another object, known as the subject, typically used in event-driven programming.
- **Q:** How would you implement a Singleton pattern in JavaScript?
    - **A:** A Singleton pattern can be implemented by creating a class or object that restricts instantiation to a single instance, often using a closure to manage the single instance.

5.1 Control Structures

- **Q:** What is the difference between == and ===?
    - **A:** == checks for equality after performing type coercion, while === checks for both value and type equality without performing type coercion.
- **Q:** How does the switch statement work?

    - **A:** The switch statement evaluates an expression, matches its value against multiple cases, and executes the block of code associated with the first matching case. If no cases match, the default block (if present) is executed.

- **Q:** What is the purpose of the break statement in loops?
    - **A:** The break statement terminates the loop or switch statement and transfers control to the statement following the loop or switch.

5.2 Type Conversion

- **Q:** What is type coercion in JavaScript?
    - **A:** Type coercion is the automatic or implicit conversion of values from one data type to another, such as converting a string to a number in a comparison.
- **Q:** How can you explicitly convert a value to a number in JavaScript?
    - **A:** You can use Number(), the unary plus +, parseInt(), or parseFloat() to explicitly convert a value to a number.
- **Q:** What is the difference between null and undefined?
    - **A:** undefined is a variable that has been declared but not assigned a value. null is an assignment value representing the intentional absence of any object value.

6.1 Arrays and Objects

- **Q:** What are the different ways to create an array in JavaScript?
    - **A:** You can create an array using array literals [], the Array constructor, or the Array.of() method.

- **Q:** How do you clone an object in JavaScript?
    - **A:** You can clone an object using Object.assign(), the spread operator {...obj}, or using JSON.parse(JSON.stringify(obj)) for a deep clone.
- **Q:** What are array methods like map, filter, and reduce?
    - **A:**
        - map: Creates a new array by applying a function to each element of the original array.
        - filter: Creates a new array with elements that pass the test implemented by the provided function.
        - reduce: Reduces the array to a single value by executing a reducer function on each element of the array.

6.2 DOM Manipulation

- **Q:** How do you select an element by ID, class, and tag name in the DOM?
    - **A:**
        - By ID: document.getElementById('id')
        - By class: document.getElementsByClassName('className')
        - By tag name: document.getElementsByTagName('tagName')

- **Q:** What is event delegation in JavaScript?
    - **A:** Event delegation is a technique where a single event listener is added to a parent element to manage events for multiple child elements. The event bubbles up from the child to the parent, allowing the parent to handle the event.
- **Q:** How do you add and remove a class from an element in JavaScript?
    - **A:** You can add a class using element.classList.add('className') and remove it using element.classList.remove('className').

7.1 Error Handling

- **Q:** What is the purpose of try...catch in JavaScript?
    - **A:** try...catch is used to handle errors in JavaScript. The try block contains code that might throw an error, and the catch block contains code to handle the error if one occurs.
- **Q:** How do you create a custom error in JavaScript?
    - **A:** You can create a custom error by extending the Error class:

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


- **Q:** What is the finally block in error handling?
    - **A:** The finally block contains code that will run after the try and catch blocks, regardless of whether an error occurred.

7.2 Promises and Async/Await

- **Q:** What happens if you don’t handle a promise rejection?
    - **A:** If a promise rejection is not handled, it will result in an unhandled promise rejection, which can cause issues in the application, such as crashes or unintended behavior.
- **Q:** How do you handle multiple promises in JavaScript?
    - **A:** You can handle multiple promises using Promise.all(), Promise.race(), Promise.allSettled(), or Promise.any() depending on the desired behavior.
- **Q:** Can you explain how async and await work together?
    - **A:** The async keyword is used to define an asynchronous function, which returns a promise. The await keyword is used to pause the execution of the async function until the promise is resolved or rejected, allowing you to write asynchronous code in a synchronous style.

7.3 Advanced Patterns and Techniques

- **Q:** What is memoization in JavaScript?
    - **A:** Memoization is a technique used to optimize function calls by caching the results of expensive function calls and returning the cached result when the same inputs occur again.

- **Q:** How can you debounce a function in JavaScript?
    - **A:** Debouncing is a technique to ensure that a function is not called too frequently. It can be implemented as follows:

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

- **Q:** What is throttling, and how do you implement it?
    - **A:** Throttling ensures that a function is only called once within a specified period, even if it is invoked multiple times. It can be implemented as follows:

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


7.4 Module Systems

- **Q:** What are ES6 modules?
    - **A:** ES6 modules are a standardized module system in JavaScript that allows you to import and export functionalities between different JavaScript files using the import and export keywords.
- **Q:** How do you export multiple values in ES6?
    - **A:** You can export multiple values using named exports:

```
export const value1 = 'Value1';
export const value2 = 'Value2';
```

- **Q:** What is the difference between default export and named export?
    - **A:** Default export allows you to export a single value from a module, which can be imported without curly braces. Named exports allow you to export multiple values, and they must be imported using curly braces.

8.1 Advanced JavaScript Concepts

- **Q:** What is a Proxy in JavaScript?
    - **A:** A Proxy is an object that allows you to define custom behavior for fundamental operations like property lookup, assignment, enumeration, function invocation, etc. It is used to intercept and redefine operations for an object.
- **Q:** What are WeakMap and WeakSet?
    - **A:** WeakMap and WeakSet are collections that hold weak references to their keys or values, meaning that if there are no other references to an object stored in them, it can be garbage collected.
- **Q:** How does JavaScript handle asynchronous iteration?
    - **A:** Asynchronous iteration in JavaScript is handled using the for await...of loop, which allows you to iterate over async iterators that return promises.

8.2 Performance Optimization

- **Q:** What are some techniques to improve performance in JavaScript applications?
    - **A:** Techniques include:
        - Minimizing DOM access and manipulation
        - Debouncing and throttling functions
        - Using lazy loading for images and resources
        - Optimizing loops and avoiding deep nesting
        - Caching results of expensive operations

- **Q:** How do you optimize large-scale data processing in JavaScript?
    - **A:** You can optimize by:
        - Using Web Workers for background processing
        - Employing streaming and chunking for large data sets
        - Leveraging libraries like lodash for efficient data manipulation
        - Using proper data structures like Map, Set, etc.
- **Q:** What is code splitting in JavaScript?
    - **A:** Code splitting is a technique used in module bundlers like Webpack to split the codebase into smaller chunks that are loaded on demand, reducing the initial load time of the application.

8.3 Security

- **Q:** What are common security vulnerabilities in JavaScript applications?
    - **A:** Common vulnerabilities include:
        - Cross-Site Scripting (XSS)
        - Cross-Site Request Forgery (CSRF)
        - SQL Injection
        - Clickjacking

        - Insecure handling of authentication and authorization
- **Q:** How do you prevent XSS attacks in JavaScript?
    - **A:** You can prevent XSS attacks by:
        - Validating and sanitizing user input
        - Using content security policies (CSP)
        - Avoiding inline JavaScript
        - Encoding data before rendering in HTML
- **Q:** How do you ensure secure handling of authentication tokens in JavaScript?
    - **A:** Secure handling of authentication tokens involves:
        - Storing tokens in secure, HTTP-only cookies
        - Using secure and same-site flags for cookies
        - Avoiding storing tokens in local storage or session storage
        - Implementing token expiration and rotation strategies

9.1 Memory Management

- **Q:** What is memory leak in JavaScript, and how can you prevent it?
    - **A:** A memory leak occurs when memory that is no longer needed is not released, causing the application to use more memory over time. This can be prevented by:

        - Removing event listeners when they are no longer needed.
        - Avoiding global variables and closures that hold references to unused objects.
        - Managing DOM references properly and removing detached DOM nodes.
- **Q:** How does garbage collection work in JavaScript?
    - **A:** JavaScript uses an automatic garbage collection mechanism that tracks object references. When an object is no longer referenced, the garbage collector can reclaim the memory. The two main strategies are:
        - **Mark-and-Sweep:** Objects are marked as "reachable" if they can be accessed from root objects. Unreachable objects are collected.
        - **Reference Counting:** Each object has a counter to track references. If the counter drops to zero, the object is collected.
- **Q:** What are weak references, and how do they differ from strong references?
    - **A:** Weak references allow the garbage collector to collect the object even if it is still referenced, unlike strong references that prevent the object from being collected. WeakMap and WeakSet are examples of weak reference collections.

9.2 Closures and Scoping

- **Q:** How do closures work under the hood in JavaScript?
    - **A:** Closures are functions that retain access to their lexical scope even when the function is executed outside that scope. Under the hood, when a function is created, it keeps a reference to its outer

scope, forming a closure. This allows the function to access variables from its outer scope even after the outer function has returned.

- **Q:** What are the implications of using closures with asynchronous code?
    - **A:** When closures are used with asynchronous code, care must be taken to ensure that the correct variable values are captured. For example, in a loop, if closures are created inside the loop, they may all reference the same variable due to the loop's scope. Using let or an IIFE can mitigate this issue.
- **Q:** How do lexical scoping and dynamic scoping differ in JavaScript?
    - **A:** JavaScript uses lexical scoping, where the scope of a variable is determined by its position in the source code. In dynamic scoping, the scope of a variable is determined at runtime based on the call stack. JavaScript's lexical scoping ensures that the function's scope chain is determined during the function definition, not during execution.

9.3 Asynchronous Patterns

- **Q:** What is an async iterator, and how does it differ from a regular iterator?
    - **A:** An async iterator is an object that defines an asynchronous iteration protocol, allowing you to loop over asynchronous data sources using for await...of. Unlike regular iterators, async iterators return promises that resolve to the next value.
- **Q:** How does the event loop handle promises and async/await?
    - **A:** The event loop handles promises by placing the then callbacks in the microtask queue, which is processed after the current task but before the next task. async/await is syntactic sugar for handling promises, pausing the execution until the promise is resolved or rejected. The await expression does not block the main thread; it allows the event loop to continue processing other tasks.

- **Q:** Can you explain what microtasks and macrotasks are in JavaScript?
    - **A:**
        - **Microtasks:** These are tasks that are queued in the microtask queue, such as promise callbacks. Microtasks are processed after the currently executing task and before the next task in the macrotask queue.
        - **Macrotasks:** These are tasks that are queued in the macrotask queue, such as setTimeout, setInterval, and I/O operations. The event loop processes one macrotask at a time and then processes all microtasks before proceeding to the next macrotask.

9.4 Advanced Function Techniques

- **Q:** What is function currying, and how is it implemented in JavaScript?
    - **A:** Function currying is the process of transforming a function that takes multiple arguments into a sequence of functions, each taking a single argument. It can be implemented as follows:

```
function curry(func) {
  return function curried(...args) {
    if (args.length >= func.length) {
      return func.apply(this, args);
    } else {
      return function(...nextArgs) {
```

```
            return curried.apply(this, args.concat(nextArgs));
        };
    }
  };
}
```

- **Q:** How does function composition work, and how can you compose multiple functions?
    - **A:** Function composition is the process of combining multiple functions into a single function, where the output of one function becomes the input of the next. It can be implemented as:

```
const compose = (...funcs) => (value) => funcs.reduceRight((acc, func) => func(acc), value);
```

- **Q:** What are higher-order functions, and can you provide an example?
    - **A:** Higher-order functions are functions that take other functions as arguments or return functions as results. An example is the map function:

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

const doubled = numbers.map(num => num * 2);
```

9.5 Advanced Object-Oriented Programming

- **Q:** How do you achieve method chaining in JavaScript?
    - **A:** Method chaining is achieved by returning the current object (this) from each method, allowing subsequent method calls to be chained. For example:


```
class Calculator {
  constructor(value = 0) {
    this.value = value;
  }
  add(num) {
    this.value += num;
    return this;
  }
  multiply(num) {
    this.value *= num;
    return this;
  }
  getResult() {
    return this.value;
  }
}
```

```
const result = new Calculator().add(5).multiply(2).getResult(); // 10
```

- **Q:** What are mixins, and how are they used in JavaScript?
    - **A:** Mixins are a way to add reusable functionality to classes. They allow multiple inheritance of behavior by mixing functions or properties into a class. An example is:

```
const CanFly = (superclass) => class extends superclass {
  fly() {
    console.log('Flying');
  }
};
const CanSwim = (superclass) => class extends superclass {
  swim() {
    console.log('Swimming');
  }
};
class Animal {}
class Duck extends CanFly(CanSwim(Animal)) {}
const duck = new Duck();
```



```
duck.fly(); // Flying
duck.swim(); // Swimming
```

- **Q:** How do you implement private methods in JavaScript classes?
    - **A:** Private methods can be implemented using the # syntax in classes:

```
class MyClass {
  #privateMethod() {
    console.log('This is a private method');
  }
  publicMethod() {
    this.#privateMethod();
  }
}
const instance = new MyClass();
instance.publicMethod(); // This is a private method
// instance.#privateMethod(); // SyntaxError: Private field '#privateMethod' must be declared in an enclosing class
```

9.6 Advanced Functional Programming

- **Q:** What is lazy evaluation, and how is it implemented in JavaScript?
    - **A:** Lazy evaluation is a strategy where the evaluation of expressions is delayed until their value is actually needed. It can be implemented using generators:

```
function* lazySequence() {
  yield 1;
  yield 2;
  yield 3;
}
const sequence = lazySequence();
console.log(sequence.next().value); // 1
console.log(sequence.next().value); // 2
console.log(sequence.next().value); // 3
```

- **Q:** What are pure functions, and why are they important in functional programming?
    - **A:** Pure functions are functions that return the same output for the same input and have no side effects (they don't modify external state). They are important because they make reasoning about code easier and enable optimizations like memoization.
- **Q:** Can you explain transducers in JavaScript?

    - **A:** Transducers are a functional programming technique that allows you to compose data transformation functions (like map, filter) without creating intermediate collections. They are implemented by combining the transformation functions into a single function that is applied in a reduce operation.

9.7 Proxies and Metaprogramming

- **Q:** How do you use a Proxy to intercept and modify operations on an object?
    - **A:** A Proxy can be used to define custom behavior for operations on an object by passing a handler object with traps (methods) like get, set, has, etc. For example:

```
const target = { message: 'Hello, World!' };
const handler = {
  get(target, property) {
    if (property === 'message') {
      return 'Intercepted: ' + target[property];
    }
    return target[property];
  }
};
```

```
const proxy = new Proxy(target, handler);

console.log(proxy.message); // Intercepted: Hello, World!
```

- **Q:** What is the Reflect API, and how does it relate to Proxies?
    - **A:** The Reflect API provides a set of static methods that mirror the behavior of fundamental operations on objects (like get, set, deleteProperty). It is often used within Proxy handlers to perform default operations, enabling the handler to focus on the custom logic.
- **Q:** How can you create a dynamic class using the new Function() constructor in JavaScript?
    - **A:** The new Function() constructor allows you to dynamically create a class by passing a string of code that defines the class. This is useful for metaprogramming tasks where classes need to be generated at runtime:

```
const MyClass = new Function('name', `
  this.name = name;
  this.sayHello = function() {
    console.log('Hello, ' + this.name);
  };
`);
const instance = new MyClass('World');
instance.sayHello(); // Hello, World
```


9.8 Advanced Error Handling

- **Q:** How do you implement custom error classes in JavaScript?
    - **A:** Custom error classes can be implemented by extending the built-in Error class:

```
class CustomError extends Error {
  constructor(message) {
    super(message);
    this.name = 'CustomError';
  }
}
try {
  throw new CustomError('Something went wrong!');
} catch (e) {
  console.error(e.name + ': ' + e.message); // CustomError: Something went wrong!
}
```

- **Q:** What is stack unwinding, and how does it occur in JavaScript?

- **A:** Stack unwinding is the process where the runtime traverses the call stack to find an appropriate catch block for an exception. In JavaScript, when an error is thrown, the stack unwinds until a catch block is found. If no catch block is found, the program terminates and an error is reported.

- **Q:** How do you handle asynchronous errors in JavaScript?
  - **A:** Asynchronous errors can be handled using:
    - **Promises:** By chaining .catch() to handle rejections.
    - **async/await:** By wrapping the code in a try...catch block.
    - **Event Listeners:** By attaching an error event listener to handle errors from asynchronous operations.

9.9 Advanced Promises and Async/Await

- **Q:** How do you handle multiple asynchronous operations that depend on each other?
  - **A:** You can handle multiple asynchronous operations that depend on each other using async/await or promise chaining. With async/await:

```
async function fetchData() {
  const user = await getUser();
  const orders = await getOrders(user.id);
  return orders;
}
```


- **Q:** How can you run multiple asynchronous operations in parallel and wait for all of them to finish?
  - **A:** Use Promise.all() to run multiple asynchronous operations in parallel and wait for all of them to finish. Eg: **const [result1, result2, result3] = await Promise.all([asyncOp1(), asyncOp2(), asyncOp3()]);**

- **Q:** What is the difference between Promise.all, Promise.any, Promise.race, and Promise.allSettled?
  - **A:**
    - **Promise.all:** Resolves when all promises are resolved or rejects when any promise is rejected.
    - **Promise.any:** Resolves when any one of the promises resolves; rejects if all promises reject.
    - **Promise.race:** Resolves or rejects as soon as any one of the promises resolves or rejects.
    - **Promise.allSettled:** Resolves after all promises have settled (either resolved or rejected) without short-circuiting.

- **Q:** What is an async generator function, and how is it used?
  - **A:** An async generator function is a function that can yield asynchronous values using the async function* syntax. It allows iterating over asynchronous data sources:

```
async function* asyncGenerator() {
  yield await Promise.resolve(1);
  yield await Promise.resolve(2);
```

```
  yield await Promise.resolve(3);
}
for await (const value of asyncGenerator()) {
  console.log(value); // 1, 2, 3
}
```

- **Q:** How can you implement retry logic with promises?
  - **A:** You can implement retry logic by recursively calling a function that returns a promise, with a condition to retry if the promise is rejected:

```
function retryAsyncOperation(fn, retries = 3) {
  return fn().catch(err => {
    if (retries === 0) throw err;
    return retryAsyncOperation(fn, retries - 1);
  });
}
retryAsyncOperation(() => fetchData()).then(console.log).catch(console.error);
```

9.10 Advanced Event Handling

- **Q:** What is event delegation, and how does it improve performance?

  - **A:** Event delegation is a technique where you attach a single event listener to a parent element to handle events for all of its child elements. This improves performance by reducing the number of event listeners attached to the DOM:

```
document.querySelector('#parent').addEventListener('click', function(event) {
  if (event.target.matches('.child')) {
    console.log('Child element clicked:', event.target);
  }
});
```

- **Q:** How does the capturing and bubbling phase work in JavaScript event handling?
  - **A:** Event handling in JavaScript occurs in two phases:
    - **Capturing Phase:** The event travels from the root to the target element, allowing event listeners to intercept the event before it reaches the target.
    - **Bubbling Phase:** The event bubbles back up from the target element to the root, allowing event listeners on parent elements to handle the event.

We can control which phase an event listener should respond to by setting the capture option to true:

```
element.addEventListener('click', handler, { capture: true });
```

- **Q:** What is the once option in event listeners, and when would you use it?

- **A:** The once option allows an event listener to be automatically removed after it is invoked once. This is useful for one-time events like initializing a feature or tracking the first user interaction:

element.addEventListener('click', handler, { once: true });

- **Q:** How can you implement custom events in JavaScript?
  - **A:** Custom events can be implemented using the CustomEvent constructor and dispatched using dispatchEvent:

const customEvent = new CustomEvent('myEvent', { detail: { message: 'Hello World!' } });

element.addEventListener('myEvent', function(event) {

console.log(event.detail.message); // Hello World!

});

element.dispatchEvent(customEvent);

9.11 Advanced RegExp (Regular Expressions)

- **Q:** How do you use lookahead and lookbehind assertions in JavaScript regular expressions?
  - **A:** Lookahead and lookbehind assertions allow you to match patterns based on what follows or precedes a certain position without including them in the match:
    - **Lookahead:** (?=pattern) matches if pattern follows the current position.
    - **Negative Lookahead:** (?!pattern) matches if pattern does not follow the current position.
    - **Lookbehind:** (?<=pattern) matches if pattern precedes the current position.
    - **Negative Lookbehind:** (?<!pattern) matches if pattern does not precede the current position.

Example:

const str = 'foo123bar';

console.log(str.match(/\d+(?=bar)/)); // ["123"]

console.log(str.match(/(?<=foo)\d+/)); // ["123"]

- **Q:** What are named capturing groups in regular expressions, and how do you use them?
  - **A:** Named capturing groups allow you to assign names to capture groups in regular expressions, making it easier to reference matched groups by name:

const regex = /(?<protocol>https?):\/\/(?<domain>\w+\.\w+)/;

const result = regex.exec('https://example.com');

console.log(result.groups.protocol); // "https"

console.log(result.groups.domain); // "example.com"

- **Q:** How do you use the replace() method with a function in JavaScript to modify matched substrings?
  - **A:** The replace() method can take a function as the second argument, allowing you to dynamically generate replacement strings based on matched substrings:

const str = 'hello world';

const result = str.replace(/(\w+)/g, (match, p1) => p1.toUpperCase());

```
console.log(result); // "HELLO WORLD"
```

9.12 Advanced Error Handling

- **Q:** How do you handle uncaught exceptions in JavaScript?
    - **A:** Uncaught exceptions can be handled using window.onerror or process.on('uncaughtException') in Node.js:

```
window.onerror = function(message, source, lineno, colno, error) {
  console.error('Uncaught exception:', message);
};

// Node.js
process.on('uncaughtException', (error) => {
  console.error('Uncaught exception:', error);
});
```

- **Q:** How do you create a global error handler for promises in JavaScript?
    - **A:** Global promise rejections can be handled using window.onunhandledrejection or process.on('unhandledRejection') in Node.js:

```
window.onunhandledrejection = function(event) {
  console.error('Unhandled rejection:', event.reason);
};

// Node.js
process.on('unhandledRejection', (reason, promise) => {
  console.error('Unhandled rejection at:', promise, 'reason:', reason);
});
```


9.13 Advanced Object Manipulation

- **Q:** How do you create an immutable object in JavaScript?
    - **A:** An immutable object can be created using Object.freeze() to prevent modifications to the object's properties:

```
const obj = Object.freeze({ name: 'Alice' });
obj.name = 'Bob'; // No effect, name remains 'Alice'
```

- **Q:** What is a shallow copy vs. a deep copy, and how do you create each in JavaScript?
    - **A:**

- **Shallow Copy:** A shallow copy copies the object's properties but does not clone nested objects. You can create a shallow copy using Object.assign() or the spread operator:

```
const obj = { name: 'Alice', address: { city: 'Wonderland' } };

const shallowCopy = { ...obj };

shallowCopy.address.city = 'New York'; // Affects the original object
```

- **Deep Copy:** A deep copy clones the object along with all nested objects. You can create a deep copy using JSON.parse(JSON.stringify(obj)) or libraries like Lodash:

```
const deepCopy = JSON.parse(JSON.stringify(obj));

deepCopy.address.city = 'New York'; // Does not affect the original object
```

- **Q:** How do you merge multiple objects with nested properties in JavaScript?
  - **A:** You can merge objects with nested properties using a recursive function or a library like Lodash:

```
const mergeObjects = (target, ...sources) => {
  sources.forEach(source => {
    Object.keys(source).forEach(key => {
      if (typeof source[key] === 'object') {
        if (!target[key]) target[key] = {};
        mergeObjects(target[key], source[key]);
      } else {
        target[key] = source[key];
      }
    });
  });
  return target;
};

const obj1 = { a: 1, b: { x: 1 } };

const obj2 = { b: { y: 2 }, c: 3 };

const result = mergeObjects({}, obj1, obj2); // { a: 1, b: { x: 1, y: 2 }, c: 3 }
```


9.14 Advanced Functionality with Closures

- **Q:** How do closures work in JavaScript, and how can they be used to create private variables?
  - **A:** A closure is a function that captures variables from its lexical scope, even after the function that created them has finished executing. Closures can be used to create private variables by returning functions that can access these captured variables:

```
function createCounter() {
```

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

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

- **Q:** What is the module pattern in JavaScript, and how does it leverage closures?
  - **A:** The module pattern is a design pattern that encapsulates private variables and functions using closures, exposing only what is necessary:

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

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

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

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


- **Q:** How can you simulate private methods in a JavaScript class?
  - **A:** Private methods in JavaScript classes can be simulated using closures or by using the # syntax (private fields/methods):

```
class MyClass {
  #privateMethod() {
```

```
    console.log('This is a private method');
  }

  publicMethod() {
    this.#privateMethod();
  }
}
const instance = new MyClass();
instance.publicMethod(); // This is a private method
```

9.15 Advanced Memory Management

- **Q:** What are memory leaks in JavaScript, and how can they occur?
  - **A:** Memory leaks occur when memory that is no longer needed is not released, leading to increased memory usage over time. Common causes include:
    - **Global Variables:** Unintentionally leaving large objects in the global scope.
    - **Closures:** Keeping references to variables in closures even after they are no longer needed.
    - **Event Listeners:** Failing to remove event listeners from DOM elements.
    - **Timers:** Forgetting to clear intervals or timeouts.

- **Q:** How can you detect and fix memory leaks in a JavaScript application?
  - **A:** Memory leaks can be detected using tools like Chrome DevTools' Memory tab. To fix them:
    - **Remove unnecessary global variables.**
    - **Ensure closures are used appropriately, and avoid retaining references unnecessarily.**
    - **Remove event listeners when they are no longer needed.**
    - **Clear timers using clearTimeout() or clearInterval().**
- **Q:** What is the role of the garbage collector in JavaScript, and how does it work?
  - **A:** The garbage collector in JavaScript automatically manages memory allocation and deallocation by identifying and reclaiming memory that is no longer reachable. It typically uses algorithms like **Mark-and-Sweep** to identify unreachable objects.

9.16 Advanced Object-Oriented Programming

- **Q:** How do you implement mixins in JavaScript?
  - **A:** Mixins are a way to add functionality to classes by copying properties and methods from one or more objects into a class. This can be done using Object.assign():

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

```
  }
};
class Person {
  constructor(name) {
    this.name = name;
  }
}
Object.assign(Person.prototype, mixin);
const person = new Person('Alice');
person.greet(); // Hello, Alice
```

- **Q:** What is polymorphism in JavaScript, and how is it achieved?
  - **A:** Polymorphism allows objects to be treated as instances of their parent class, enabling a single function to operate on different types of objects. It is achieved through method overriding:

```
class Animal {
  speak() {
    console.log('Animal sound');
  }
}
```



```
class Dog extends Animal {
  speak() {
    console.log('Bark');
  }
}
class Cat extends Animal {
  speak() {
    console.log('Meow');
  }
}
const animals = [new Dog(), new Cat()];
animals.forEach(animal => animal.speak()); // Bark, Meow
```

- **Q:** What is method chaining in JavaScript, and how can it be implemented?
  - **A:** Method chaining allows multiple methods to be called on the same object sequentially. It is implemented by returning this from each method:

```
class Calculator {
```

```
    constructor(value = 0) {
        this.value = value;
    }
    add(value) {
        this.value += value;
        return this;
    }
    subtract(value) {
        this.value -= value;
        return this;
    }
multiply(value) {
        this.value *= value;
        return this;
    }
    getResult() {
        return this.value;
    }
}
const result = new Calculator()
    .add(10)
    .subtract(2)
    .multiply(3)
    .getResult(); // 24
```


9.17 Advanced Asynchronous Patterns

- **Q:** What is a Promise.allSettled and how does it differ from Promise.all?
    - **A:** Promise.allSettled waits for all promises to settle (either resolved or rejected) and returns an array of objects representing the outcome of each promise. Unlike Promise.all, it does not short-circuit on rejection:

```
const promises = [
    Promise.resolve(1),
    Promise.reject('Error'),
    Promise.resolve(2)
];
```

```
Promise.allSettled(promises).then(results => console.log(results));
// [
//   { status: 'fulfilled', value: 1 },
//   { status: 'rejected', reason: 'Error' },
//   { status: 'fulfilled', value: 2 }
// ]
```

- **Q:** How do you implement a throttling function in JavaScript?
  - **A:** Throttling ensures that a function is not called more frequently than a specified interval. It can be implemented using a timer:

```
function throttle(func, delay) {
  let lastCall = 0;
  return function(...args) {
    const now = Date.now();
    if (now - lastCall >= delay) {
      lastCall = now;
      func.apply(this, args);
    }
  };
}
window.addEventListener('resize', throttle(() => console.log('Resized!'), 200));
```


- **Q:** What is the difference between throttling and debouncing?
  - **A: Throttling** limits the execution of a function to once every specified time interval, while **debouncing** delays the execution of a function until after a specified time has passed since the last call. Debouncing is useful for events like keystrokes, where we want to wait until the user stops typing before processing the input.

9.18 Advanced Array Methods

- **Q:** How does the Array.prototype.reduce() method work, and what are its use cases?
  - **A:** The reduce() method executes a reducer function on each element of the array, resulting in a single output value. It is commonly used for summing values, flattening arrays, or accumulating results:

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

- **Q:** How can you implement a custom Array.prototype.map() using reduce()?

- **A:** A custom map() can be implemented using reduce() by accumulating the transformed elements:

```
const customMap = (arr, callback) =>
  arr.reduce((acc, current, index, array) => {
    acc.push(callback(current, index, array));
    return acc;
  }, []);
const doubled = customMap([1, 2, 3], x => x * 2);
console.log(doubled); // [2, 4, 6]
```

- **Q:** How do you implement the flatMap() method in JavaScript?
  - **A:** The flatMap() method maps each element using a mapping function, then flattens the result by one level:

```
const arr = [1, 2, 3];
const flatMapped = arr.flatMap(x => [x, x * 2]);
console.log(flatMapped); // [1, 2, 2, 4, 3, 6]
```