Javascript Q & A

1. What is a closure in JavaScript and how does it work?

A closure is a function that has access to its outer scope's variables, even when the outer function has finished execution. This allows the inner function to "remember" the variables and use them as needed.

Example:

```
function outer() {
    let count = 0;
    function inner() {
        count++;
        console.log(count);
    }
    return inner;
}

const counter = outer();
counter(); // 1
counter(); // 2
```

2. How do you use closures to create private variables?

Closures can encapsulate variables, making them accessible only through returned functions, thus creating private variables.

Example:

```
function bankAccount(initialBalance) {
    let balance = initialBalance;
    return {
        deposit: function(amount) {
```

```
            balance += amount;
        },
        getBalance: function() {
            return balance;
        }
    };
}

const account = bankAccount(100);
account.deposit(50);
console.log(account.getBalance()); // 150
```

3. What is lexical scope in JavaScript?

Lexical scope means that the accessibility of variables is determined by their placement in the source code. Inner functions have access to the variables of outer functions in which they are nested.

Example:

```
function outer() {
    const name = "John";
    function inner() {
        console.log(name); // Accesses name from outer scope
    }
    inner();
}
outer();
```

4. How do closures help in data privacy?

Closures help by encapsulating data, making variables accessible only through specific functions, thus preventing direct external modification.

Example:

```
function createPerson(name) {
    let age = 0;
    return {
        getName: () => name,
```

```
        getAge: () => age,
        setAge: (newAge) => age = newAge
    };
}

const person = createPerson("Alice");
console.log(person.getName()); // Alice
person.setAge(30);
console.log(person.getAge()); // 30
```

5. Explain the difference between var, let, and const in terms of scope.

- **var**: Function scope; accessible throughout the function.
- **let** and **const**: Block scope; accessible only within the block they're declared.

Example:

```
function scopeTest() {
    if (true) {
        var a = 10; // Function scope
        let b = 20; // Block scope
        const c = 30; // Block scope
    }
    console.log(a); // 10
    console.log(b); // ReferenceError
    console.log(c); // ReferenceError
}
scopeTest();
```

6. Explain how Array.reduce() works with an example.

Reduce applies a function to each element, accumulating a result. It takes an initial value and processes each element to produce a single output.

Example:

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

7. How would you implement a custom reduce() method?

Create a function that iterates over the array, applying a callback to accumulate results.

Example:

```
function customReduce(array, callback, initialValue) {
    let accumulator = initialValue;
    for (let i = 0; i < array.length; i++) {
        accumulator = callback(accumulator, array[i], i, array);
    }
    return accumulator;
}

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

8. What are the key differences between map() and forEach()?

- **map()**: Returns a new array with transformed elements.
- **forEach()**: Executes a function for each element without returning a new array.

Example:

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

9. How do you remove duplicates from an array?

Use a Set to automatically remove duplicates, then convert back to an array.

Example:

```
const duplicates = [1, 2, 2, 3, 4, 4];
const unique = [...new Set(duplicates)]; // [1, 2, 3, 4]
```

10. Explain the difference between slice() and splice().

- **slice()**: Returns a new array without modifying the original.
- **splice()**: Modifies the original array by removing or adding elements.

Example:

```
const array = [1, 2, 3, 4];
const sliced = array.slice(1, 3); // [2, 3]
const spliced = array.splice(1, 2, 5); // [1, 5, 4]
console.log(array); // [1, 5, 4]
```

11. What is a Promise and what problem does it solve?

A Promise is an object representing the eventual completion (or failure) of an asynchronous operation. It solves the problem of "callback hell" by providing a cleaner, more readable way to handle asynchronous code.

Example:

```
const fetchData = new Promise((resolve, reject) => {
    setTimeout(() => {
        resolve("Data fetched successfully!");
    }, 1000);
});

fetchData
    .then(data => console.log(data)) // "Data fetched successfully!"
    .catch(error => console.error(error));
```

12. How do you implement a custom Promise?

A custom Promise can be implemented by creating a class with `resolve` and `reject` methods and maintaining the state (`pending`, `fulfilled`, or `rejected`).

Example:

```
class CustomPromise {
  constructor(executor) {
    this.state = "pending";
    this.value = undefined;
    this.callbacks = [];

    const resolve = (value) => {
      if (this.state === "pending") {
        this.state = "fulfilled";
        this.value = value;
        this.callbacks.forEach(callback => callback(value));
      }
    };

    const reject = (reason) => {
      if (this.state === "pending") {
        this.state = "rejected";
        this.value = reason;
      }
    };

    executor(resolve, reject);
  }

  then(onFulfilled) {
    if (this.state === "fulfilled") {
      onFulfilled(this.value);
    } else {
      this.callbacks.push(onFulfilled);
    }
  }
}

// Example usage:
const promise = new CustomPromise((resolve, reject) => {
  setTimeout(() => resolve("Custom Promise Resolved!"), 1000);
```

```
});

promise.then(data => console.log(data)); // "Custom Promise Resolved!"
```

13. Explain Promise.all() and its use cases.

`Promise.all()` takes an array of Promises and resolves when all of them are resolved. If any Promise is rejected, it immediately rejects.

Example:

```
const promise1 = Promise.resolve(10);
const promise2 = Promise.resolve(20);
const promise3 = Promise.resolve(30);

Promise.all([promise1, promise2, promise3])
  .then(results => console.log(results)) // [10, 20, 30]
  .catch(error => console.error(error));
```

14. How does Promise.race() differ from Promise.any()?

- `Promise.race()` resolves or rejects as soon as the first Promise settles (either resolved or rejected).
- `Promise.any()` resolves as soon as the first Promise is fulfilled. If all Promises are rejected, it throws an `AggregateError`.

Example:

```
const promise1 = new Promise((resolve) => setTimeout(resolve, 100, "Firs
t"));
const promise2 = new Promise((resolve) => setTimeout(resolve, 200, "Seco
nd"));

Promise.race([promise1, promise2]).then(result => console.log(result)); // "F
irst"
Promise.any([Promise.reject("Error"), promise2]).then(result => console.log
(result)); // "Second"
```

15. What is the purpose of Promise.finally()?

`Promise.finally()` is used to execute code after a Promise is settled (resolved or rejected), regardless of the outcome.

Example:

```
fetch("https://api.example.com/data")
  .then(response ⇒ console.log("Data fetched"))
  .catch(error ⇒ console.error("Error occurred"))
  .finally(() ⇒ console.log("Cleanup actions"));
```

16. How does `this` work in JavaScript?

The value of `this` depends on how a function is called:

- In a method, `this` refers to the object.
- In a regular function, `this` refers to the global object (`window` in browsers).
- In strict mode, `this` is `undefined` in regular functions.

Example:

```
const obj = {
  name: "Alice",
  greet: function() {
    console.log(this.name);
  }
};

obj.greet(); // "Alice"
```

17. What are the different ways to bind `this`?

- **Explicit binding**: Using `call()`, `apply()`, or `bind()`.
- **Arrow functions**: Automatically bind `this` to the surrounding context.

Example:

```
function greet() {
  console.log(this.name);
}
```

```
const person = { name: "Alice" };
greet.call(person); // "Alice"
```

18. Explain `call()`, `apply()`, and `bind()`.

- `call()`: Invokes a function with a specific `this` value and arguments passed individually.
- `apply()`: Similar to `call()`, but arguments are passed as an array.
- `bind()`: Returns a new function with `this` permanently bound.

Example:

```
function greet(greeting) {
    console.log(`${greeting}, ${this.name}`);
}

const person = { name: "Alice" };
greet.call(person, "Hello"); // "Hello, Alice"
greet.apply(person, ["Hi"]); // "Hi, Alice"
const boundGreet = greet.bind(person, "Hey");
boundGreet(); // "Hey, Alice"
```

19. What is a higher-order function?

A higher-order function is a function that takes another function as an argument or returns a function.

Example:

```
function higherOrder(fn) {
    return function(x) {
        return fn(x) * 2;
    };
}

const double = higherOrder(x => x + 1);
console.log(double(5)); // 12
```

20. How do arrow functions handle `this`?

Arrow functions do not have their own `this`. Instead, they inherit `this` from their surrounding lexical scope.

Example:

```
const obj = {
  name: "Alice",
  greet: () => console.log(this.name)
};

obj.greet(); // undefined (inherits `this` from global scope)
```

Object-Oriented JavaScript

21. How do you implement inheritance in JavaScript?

Inheritance can be implemented using `class` syntax or prototypes.

Example:

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

class Dog extends Animal {
  speak() {
    console.log(`${this.name} barks.`);
  }
}

const dog = new Dog("Rex");
dog.speak(); // "Rex barks."
```

22. What is prototypal inheritance?

Prototypal inheritance allows objects to inherit properties and methods from other objects via the prototype chain.

Example:

```
const animal = {
  speak: function() {
    console.log(`${this.name} makes a noise.`);
  }
};

const dog = Object.create(animal);
dog.name = "Rex";
dog.speak(); // "Rex makes a noise."
```

23. How do you check if an object is an instance of a class?

Use the `instanceof` operator to check if an object is an instance of a class.

Example:

```
class Animal {}
const dog = new Animal();
console.log(dog instanceof Animal); // true
```

24. What are getters and setters?

Explanation:

Getters and setters allow you to define methods that are accessed like properties.

Example:

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

```
    set name(newName) {
        this._name = newName;
    }
}

const person = new Person("Alice");
console.log(person.name); // "Alice"
person.name = "Bob";
console.log(person.name); // "Bob"
```

25. How do you prevent object modification?

Use `Object.freeze()` to make an object immutable.

Example:

```
const obj = { name: "Alice" };
Object.freeze(obj);
obj.name = "Bob"; // No effect
console.log(obj.name); // "Alice"
```

Event Handling

26. What is event bubbling?

Event bubbling is the process where an event propagates from the target element up to its ancestors.

27. How do you implement event delegation?

Event delegation involves attaching a single event listener to a parent element to handle events for its child elements.

Example:

```
document.getElementById("parent").addEventListener("click", (event) => {
    if (event.target.tagName === "BUTTON") {
        console.log("Button clicked:", event.target.textContent);
    }
});
```

28. Explain the event loop.

The event loop is a mechanism that handles asynchronous operations by continuously checking the call stack and the task queue.

29. What is the difference between capture and bubble phase?

- **Capture phase**: Event propagates from the root to the target.
- **Bubble phase**: Event propagates from the target back to the root.

30. How do you stop event propagation?

Use `event.stopPropagation()` to stop the event from propagating further.

Example:

```
document.getElementById("child").addEventListener("click", (event) => {
    event.stopPropagation();
    console.log("Child clicked");
});
```

DOM Manipulation

31. How do you efficiently add multiple elements to the DOM?

To efficiently add multiple elements, use a `DocumentFragment` to batch updates and minimize reflows and repaints.

Example:

```
const fragment = document.createDocumentFragment();
for (let i = 0; i < 5; i++) {
    const div = document.createElement("div");
    div.textContent = `Item ${i}`;
    fragment.appendChild(div);
}
document.body.appendChild(fragment);
```

32. What is the virtual DOM?

The virtual DOM is a lightweight, in-memory representation of the real DOM. It is used in libraries like React to optimize updates by comparing the virtual DOM

with the real DOM (diffing) and applying minimal changes.

33. How do you find elements with a specific class name?

Use `document.getElementsByClassName()` or `document.querySelectorAll()`.

Example:

```
const elements = document.getElementsByClassName("my-class");
console.log(elements);

const elementsQuery = document.querySelectorAll(".my-class");
console.log(elementsQuery);
```

34. How would you implement a custom querySelector?

A custom `querySelector` can be implemented using recursion to traverse the DOM tree.

Example:

```
function customQuerySelector(selector, root = document) {
    if (root.matches(selector)) return root;
    for (const child of root.children) {
        const result = customQuerySelector(selector, child);
        if (result) return result;
    }
    return null;
}

const element = customQuerySelector(".my-class");
console.log(element);
```

35. What is DOM traversal?

DOM traversal refers to navigating through the DOM tree using properties like `parentNode`, `childNodes`, `nextSibling`, and `previousSibling`.

Example:

```
const parent = document.getElementById("parent");
const firstChild = parent.firstChild;
const nextSibling = firstChild.nextSibling;
console.log(nextSibling);
```

Design Patterns

36. Explain the Singleton pattern.

The Singleton pattern ensures that a class has only one instance and provides a global point of access to it.

Example:

```
const Singleton = (function () {
  let instance;
  function createInstance() {
    return { name: "Singleton Instance" };
  }
  return {
    getInstance: function () {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
  };
})();

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

37. How do you implement the Observer pattern?

The Observer pattern allows objects (observers) to subscribe to events and get notified when the event occurs.

Example:

```
class Subject {
    constructor() {
        this.observers = [];
    }
    subscribe(observer) {
        this.observers.push(observer);
    }
    notify(data) {
        this.observers.forEach(observer => observer(data));
    }
}

const subject = new Subject();
subject.subscribe(data => console.log(`Observer 1: ${data}`));
subject.subscribe(data => console.log(`Observer 2: ${data}`));
subject.notify("Event occurred!");
```

38. What is the Module pattern?

The Module pattern is used to encapsulate private and public methods and variables, providing a clean API.

Example:

```
const Module = (function () {
    let privateVar = "I am private";
    function privateMethod() {
        console.log(privateVar);
    }
    return {
        publicMethod: function () {
            privateMethod();
        }
    };
})();

Module.publicMethod(); // "I am private"
```

39. Explain the Factory pattern.

The Factory pattern provides a way to create objects without specifying their exact class.

Example:

```
function Car(make, model) {
    this.make = make;
    this.model = model;
}

function CarFactory() {
    this.createCar = function (make, model) {
        return new Car(make, model);
    };
}

const factory = new CarFactory();
const car1 = factory.createCar("Toyota", "Corolla");
console.log(car1);
```

40. What is the Pub/Sub pattern?

The Publish/Subscribe pattern allows components to communicate indirectly through events.

Example:

```
const PubSub = {
    events: {},
    subscribe: function (event, callback) {
        if (!this.events[event]) this.events[event] = [];
        this.events[event].push(callback);
    },
    publish: function (event, data) {
        if (this.events[event]) {
            this.events[event].forEach(callback => callback(data));
        }
    }
```

```
};

PubSub.subscribe("event1", data => console.log(`Received: ${data}`));
PubSub.publish("event1", "Hello, Pub/Sub!");
```

Error Handling

41. How do you implement proper error handling?

Use `try...catch` blocks to handle errors gracefully and prevent crashes.

Example:

```
try {
    throw new Error("Something went wrong!");
} catch (error) {
    console.error(error.message);
}
```

42. What is the difference between `throw` and `return`?

- `throw`: Stops execution and propagates an error.
- `return`: Exits a function and optionally returns a value.

Example:

```
function testThrow() {
    throw new Error("Error thrown!");
}

function testReturn() {
    return "Function returned!";
}

try {
    testThrow();
} catch (error) {
    console.error(error.message);
}
```

```
console.log(testReturn());
```

43. How do `try/catch` blocks work with async code?

For async code, use `try/catch` with `async/await` to handle errors.

Example:

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

44. What are custom error classes?

Custom error classes allow you to create specific error types by extending the `Error` class.

Example:

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

try {
    throw new CustomError("This is a custom error!");
} catch (error) {
    console.error(error.name, error.message);
}
```

45. How do you handle uncaught exceptions?

Use `process.on("uncaughtException")` in Node.js or `window.onerror` in browsers.

Example (Node.js):

```
process.on("uncaughtException", (error) => {
  console.error("Uncaught Exception:", error);
});
throw new Error("Unhandled error!");
```

Performance

46. How do you implement debouncing?

Debouncing ensures a function is executed only after a specified delay.

Example:

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

const debouncedFunc = debounce(() => console.log("Debounced!"), 300);
debouncedFunc();
debouncedFunc();
```

47. What is throttling and how is it different from debouncing?

Throttling ensures a function is executed at most once in a specified interval, while debouncing delays execution until no further calls occur.

Example:

```
function throttle(func, limit) {
  let inThrottle;
  return function (...args) {
```

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

const throttledFunc = throttle(() => console.log("Throttled!"), 300);
throttledFunc();
throttledFunc();
```

48. How do you optimize JavaScript code?

- Minimize DOM manipulations.
- Use `requestAnimationFrame` for animations.
- Avoid memory leaks by cleaning up event listeners.

49. What are memory leaks and how do you prevent them?

Memory leaks occur when objects are no longer needed but are not garbage collected. Prevent them by:

- Removing event listeners.
- Avoiding global variables.

50. How do you implement infinite scrolling efficiently?

Use an `IntersectionObserver` to detect when the user scrolls near the bottom and load more content.

Example:

```
const observer = new IntersectionObserver((entries) => {
    if (entries[0].isIntersecting) {
        console.log("Load more content");
    }
});
```

```
observer.observe(document.querySelector("#scroll-anchor"));
```