Advanced JavaScript Patterns

Advanced JavaScript Questions

1. Implement a Singleton Pattern

Question: Create a Singleton class in JavaScript that restricts the instantiation of a class to a single instance.
Solution:

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

    someMethod() {
        console.log("This method belongs to the singleton instance.");
    }
}

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

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

2. Implement the Module Pattern

Question: Create a module that maintains a private variable and exposes methods to interact with it.
Solution:

```javascript
const CounterModule = (function() {
    let count = 0;
```

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

    console.log(CounterModule.increment()); // 1
    console.log(CounterModule.increment()); // 2
    console.log(CounterModule.getCount());  // 2
```

3. Create a Factory Pattern

Question: Create a factory function that generates different types of users.
Solution:

javascript

```
    function UserFactory() {
        this.createUser = function(type) {
            let user;

            if (type === "admin") {
                user = new Admin();
            } else if (type === "editor") {
                user = new Editor();
            } else {
                user = new Viewer();
            }

            user.type = type;

            return user;
        };
    }

    class Admin {
        constructor() {
            this.permissions = ["read", "write", "delete"];
        }
    }

    class Editor {
        constructor() {
            this.permissions = ["read", "write"];
        }
    }

    class Viewer {
        constructor() {
            this.permissions = ["read"];
        }
    }

    const factory = new UserFactory();
    const adminUser = factory.createUser("admin");
    console.log(adminUser.permissions); // ["read", "write", "delete"]
```

4. Implement the Observer Pattern

Question: Create a simple pub-sub system using the Observer pattern.
Solution:

javascript

```javascript
class EventEmitter {
    constructor() {
        this.events = {};
    }

    subscribe(event, listener) {
        if (!this.events[event]) {
            this.events[event] = [];
        }
        this.events[event].push(listener);
    }

    unsubscribe(event, listener) {
        if (this.events[event]) {
            this.events[event] = this.events[event].filter(l => l !== listener);
        }
    }

    emit(event, data) {
        if (this.events[event]) {
            this.events[event].forEach(listener => listener(data));
        }
    }
}

// Usage
const emitter = new EventEmitter();
const onEvent = (data) => console.log(`Received: ${data}`);

emitter.subscribe('dataReceived', onEvent);
emitter.emit('dataReceived', 'Hello, Observer!'); // Received: Hello, Observer!
```

5. Implement a Promise-Based API

Question: Create a function that returns a promise which resolves after a timeout.
Solution:

javascript

```javascript
function delayedPromise(timeout) {
    return new Promise((resolve) => {
        setTimeout(() => {
            resolve("Resolved after " + timeout + "ms");
        }, timeout);
    });
}

// Usage
delayedPromise(2000).then(console.log); // Resolved after 2000ms
```

6. Implement Debouncing

Question: Write a debounce function that limits the rate at which a function can fire.

Solution:

```javascript
function debounce(func, delay) {
    let timeoutId;

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

// Usage
const handleResize = debounce(() => {
    console.log("Window resized!");
}, 300);

window.addEventListener('resize', handleResize);
```

7. Implement Throttling

Question: Create a throttle function that limits the number of times a function can be called over time.
Solution:

```javascript
function throttle(func, limit) {
    let lastFunc;
    let lastRan;

    return function(...args) {
        if (!lastRan) {
            func.apply(this, args);
            lastRan = Date.now();
        } else {
            clearTimeout(lastFunc);
            lastFunc = setTimeout(() => {
                if (Date.now() - lastRan >= limit) {
                    func.apply(this, args);
                    lastRan = Date.now();
                }
            }, limit - (Date.now() - lastRan));
        }
    };
}

// Usage
const logScroll = throttle(() => {
    console.log("Scrolled!");
}, 1000);

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

8. Implement a Simple MVC Pattern

Question: Create a simple MVC pattern with a model, view, and controller.
Solution:

javascript

```javascript
class Model {
    constructor() {
        this.data = [];
    }

    addItem(item) {
        this.data.push(item);
        this.notifyObservers();
    }

    getItems() {
        return this.data;
    }

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

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

class View {
    constructor() {
        this.app = document.getElementById('app');
    }

    render(items) {
        this.app.innerHTML = '';
        items.forEach(item => {
            const div = document.createElement('div');
            div.innerText = item;
            this.app.appendChild(div);
        });
    }
}

class Controller {
    constructor(model, view) {
        this.model = model;
        this.view = view;

        this.model.subscribe(this);
    }

    update() {
        this.view.render(this.model.getItems());
    }

    addItem(item) {
        this.model.addItem(item);
    }
}

// Usage
const model = new Model();
const view = new View();
const controller = new Controller(model, view);

controller.addItem('Item 1');
controller.addItem('Item 2');
```

9. Create a Custom Fetch Function

Question: Write a custom fetch function that adds headers and handles errors.

Solution:

```javascript
async function customFetch(url, options = {}) {
    const defaultHeaders = {
        'Content-Type': 'application/json',
        'Accept': 'application/json'
    };

    options.headers = { ...defaultHeaders, ...options.headers };

    try {
        const response = await fetch(url, options);
        if (!response.ok) {
            throw new Error(`HTTP error! Status: ${response.status}`);
        }
        return await response.json();
    } catch (error) {
        console.error('Fetch error:', error);
        throw error;
    }
}

// Usage
customFetch('https://api.example.com/data')
    .then(data => console.log(data))
    .catch(error => console.error(error));
```

10. Implement Error Handling with Try-Catch in Async Functions

Question: Write an async function that fetches data and handles errors with try-catch.
Solution:

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

// Usage
fetchData('https://api.example.com/data');
```

11. Implement a Deep Clone Function

Question: Write a function that creates a deep clone of an object.
Solution:

```javascript
function deepClone(obj) {
    return JSON.parse(JSON.stringify(obj));
}

// Usage
const original = { name: 'Alice', age: 30, nested: { active: true } };
const clone = deepClone(original);
clone.nested.active = false;
console.log(original.nested.active); // true
```

12. Implement a Simple Event Delegation

Question: Create an event delegation pattern for a list of items.
Solution:

```javascript
document.getElementById('list').addEventListener('click', function(event) {
    if (event.target.tagName === 'LI') {
        console.log(`Item clicked: ${event.target.textContent}`);
    }
});

// HTML:
// <ul id="list">
//   <li>Item 1</li>
//   <li>Item 2</li>
//   <li>Item 3</li>
// </ul>
```

13. Create a Context API

Question: Implement a simple Context API to share data across components.
Solution:

```javascript
class Context {
    constructor(value) {
        this.value = value;
        this.subscribers = [];
    }

    subscribe(callback) {
        this.subscribers.push(callback);
    }

    update(value) {
        this.value = value;
        this.subscribers.forEach(callback => callback(this.value));
    }
}

// Usage
const themeContext = new Context('light');

themeContext.subscribe(value => {
    console.log(`Theme updated: ${value}`);
```

```
});

themeContext.update('dark'); // Theme updated: dark
```

14. Implement an Async Queue

Question: Write a simple async queue that processes tasks in order.
Solution:

javascript

```
class AsyncQueue {
    constructor() {
        this.queue = [];
        this.processing = false;
    }

    async add(task) {
        this.queue.push(task);
        this.processQueue();
    }

    async processQueue() {
        if (this.processing) return;
        this.processing = true;

        while (this.queue.length > 0) {
            const task = this.queue.shift();
            await task();
        }

        this.processing = false;
    }
}

// Usage
const queue = new AsyncQueue();

queue.add(async () => {
    console.log('Task 1 start');
    await new Promise(resolve => setTimeout(resolve, 1000));
    console.log('Task 1 end');
});

queue.add(async () => {
    console.log('Task 2 start');
    await new Promise(resolve => setTimeout(resolve, 500));
    console.log('Task 2 end');
});
```

15. Implement a Simple State Management

Question: Create a simple state management solution.
Solution:

javascript

```
class Store {
    constructor() {
        this.state = {};
        this.listeners = [];
    }
```

```
    setState(newState) {
        this.state = { ...this.state, ...newState };
        this.listeners.forEach(listener => listener(this.state));
    }

    subscribe(listener) {
        this.listeners.push(listener);
    }
}

// Usage
const store = new Store();

store.subscribe(state => {
    console.log('State changed:', state);
});

store.setState({ count: 1 });
store.setState({ count: 2 });
```

16. Implement a Middleware Function

Question: Create a simple middleware function to log actions in a store.
Solution:

javascript

```
function loggerMiddleware(store) {
    const originalDispatch = store.dispatch;

    store.dispatch = function(action) {
        console.log('Dispatching action:', action);
        originalDispatch.call(store, action);
    };
}

// Usage
const store = {
    state: {},
    dispatch(action) {
        console.log('Action dispatched:', action);
    }
};

loggerMiddleware(store);
store.dispatch({ type: 'ADD_ITEM' }); // Dispatching action: { type: 'ADD_ITEM' }
```

17. Implement a Simple Router

Question: Create a basic client-side router.
Solution:

javascript

```
class Router {
    constructor(routes) {
        this.routes = routes;
        this.currentRoute = '';
    }

    navigate(route) {
```

```
            this.currentRoute = route;
            const handler = this.routes[route];
            if (handler) {
                handler();
            } else {
                console.log('404 - Not Found');
            }
        }
    }

    // Usage
    const routes = {
        '/': () => console.log('Home'),
        '/about': () => console.log('About'),
    };

    const router = new Router(routes);
    router.navigate('/');      // Home
    router.navigate('/about');  // About
    router.navigate('/contact'); // 404 - Not Found
```

18. Implement Local Storage with a Wrapper

Question: Create a simple wrapper around local storage.
Solution:

```javascript
    class Storage {
        static set(key, value) {
            localStorage.setItem(key, JSON.stringify(value));
        }

        static get(key) {
            const value = localStorage.getItem(key);
            return value ? JSON.parse(value) : null;
        }

        static remove(key) {
            localStorage.removeItem(key);
        }
    }

    // Usage
    Storage.set('user', { name: 'Alice', age: 30 });
    console.log(Storage.get('user')); // { name: 'Alice', age: 30 }
    Storage.remove('user');
```

19. Implement a Custom Event Emitter

Question: Create a simple EventEmitter class.
Solution:

```javascript
    class EventEmitter {
        constructor() {
            this.events = {};
        }

        on(event, listener) {
            if (!this.events[event]) {
```

```
            this.events[event] = [];
        }
        this.events[event].push(listener);
    }

    emit(event, ...args) {
        if (this.events[event]) {
            this.events[event].forEach(listener => listener(...args));
        }
    }
}

// Usage
const emitter = new EventEmitter();
emitter.on('data', data => console.log('Data received:', data));
emitter.emit('data', { key: 'value' }); // Data received: { key: 'value' }
```

20. Implement a Fetch Wrapper with Retry Logic

Question: Create a fetch wrapper that retries on failure.
Solution:

javascript

```
async function fetchWithRetry(url, options = {}, retries = 3) {
    for (let i = 0; i < retries; i++) {
        try {
            const response = await fetch(url, options);
            if (!response.ok) throw new Error('Network response was not ok');
            return await response.json();
        } catch (error) {
            console.error(`Attempt ${i + 1} failed: ${error.message}`);
            if (i === retries - 1) throw error;
        }
    }
}

// Usage
fetchWithRetry('https://api.example.com/data')
    .then(data => console.log(data))
    .catch(error => console.error('Final error:', error));
```

21. Implement a Simple Undo/Redo Stack

Question: Create a class that manages a stack for undo and redo operations.
Solution:

javascript

```
class History {
    constructor() {
        this.undoStack = [];
        this.redoStack = [];
    }

    execute(command) {
        this.undoStack.push(command);
        this.redoStack = []; // Clear redo stack
        command.execute();
```

```
    }

    undo() {
        if (this.undoStack.length) {
            const command = this.undoStack.pop();
            command.undo();
            this.redoStack.push(command);
        }
    }

    redo() {
        if (this.redoStack.length) {
            const command = this.redoStack.pop();
            command.execute();
            this.undoStack.push(command);
        }
    }
}

// Usage
class Command {
    constructor(action) {
        this.action = action;
    }

    execute() {
        console.log(`Executing: ${this.action}`);
    }

    undo() {
        console.log(`Undoing: ${this.action}`);
    }
}

const history = new History();
const command1 = new Command('Action 1');
const command2 = new Command('Action 2');

history.execute(command1);
history.execute(command2);
history.undo();
history.redo();
```

22. Implement a Simple Cache

Question: Write a caching mechanism using a Map.
Solution:

```javascript
class Cache {
    constructor() {
        this.cache = new Map();
    }

    get(key) {
        return this.cache.get(key);
    }

    set(key, value) {
        this.cache.set(key, value);
    }

    clear() {
        this.cache.clear();
    }
}
```

```
// Usage
const cache = new Cache();
cache.set('a', 1);
console.log(cache.get('a')); // 1
cache.clear();
console.log(cache.get('a')); // undefined
```

23. Implement a Simple Form Validation

Question: Write a function that validates a form object.
Solution:

```javascript
function validateForm(formData) {
    const errors = {};
    if (!formData.username) {
        errors.username = 'Username is required';
    }
    if (!formData.email.includes('@')) {
        errors.email = 'Email must be valid';
    }
    return errors;
}

// Usage
const formData = { username: '', email: 'test.com' };
const validationErrors = validateForm(formData);
console.log(validationErrors); // { username: 'Username is required', email: 'Email must
be valid' }
```

24. Implement a Simple Pagination Function

Question: Write a function that paginates an array.
Solution:

```javascript
function paginate(array, pageSize, pageNumber) {
    return array.slice((pageNumber - 1) * pageSize, pageNumber * pageSize);
}

// Usage
const items = Array.from({ length: 50 }, (_, i) => i + 1);
const page = paginate(items, 10, 2);
console.log(page); // [11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
```

25. Create a Basic Fetch Wrapper with Abort

Question: Implement a fetch wrapper that allows for aborting requests.
Solution:

```javascript
function fetchWithAbort(url, options = {}, signal) {
    return fetch(url, { ...options, signal })
```

```
        .then(response => {
            if (!response.ok) throw new Error('Network response was not ok');
            return response.json();
        });
}

// Usage
const controller = new AbortController();
fetchWithAbort('https://api.example.com/data', { signal: controller.signal })
    .then(data => console.log(data))
    .catch(error => console.error('Error:', error));

// To abort
controller.abort();
```

26. Implement a Simple Timer with Promises

Question: Create a timer function that returns a promise.
Solution:

javascript

```
function timer(seconds) {
    return new Promise(resolve => {
        setTimeout(() => {
            resolve(`Timer finished after ${seconds} seconds`);
        }, seconds * 1000);
    });
}

// Usage
timer(3).then(console.log); // Timer finished after 3 seconds
```

27. Implement a Simple Async/Await Queue

Question: Create a queue that processes async tasks sequentially.
Solution:

javascript

```
class AsyncQueue {
    constructor() {
        this.tasks = [];
        this.processing = false;
    }

    add(task) {
        this.tasks.push(task);
        this.processQueue();
    }

    async processQueue() {
        if (this.processing) return;
        this.processing = true;

        while (this.tasks.length > 0) {
            const task = this.tasks.shift();
            await task();
        }

        this.processing = false;
    }
```

```
}

// Usage
const asyncQueue = new AsyncQueue();

asyncQueue.add(async () => {
    console.log('Task 1 start');
    await new Promise(resolve => setTimeout(resolve, 1000));
    console.log('Task 1 end');
});

asyncQueue.add(async () => {
    console.log('Task 2 start');
    await new Promise(resolve => setTimeout(resolve, 500));
    console.log('Task 2 end');
});
```

28. Create a Simple Memoization Function

Question: Write a memoization function to cache results of a function.
Solution:

```
javascript
```

```
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(...args);
        cache.set(key, result);
        return result;
    };
}

// Usage
const factorial = memoize(n => (n <= 1 ? 1 : n * factorial(n - 1)));
console.log(factorial(5)); // 120
```

29. Implement a Simple Command Pattern

Question: Create a command pattern to manage actions.
Solution:

```
javascript
```

```
class Command {
    constructor(action) {
        this.action = action;
    }

    execute() {
        console.log(`Executing: ${this.action}`);
    }
}

class CommandManager {
    constructor() {
```

```
        this.commands = [];
    }

    execute(command) {
        command.execute();
        this.commands.push(command);
    }
}

// Usage
const manager = new CommandManager();
const command1 = new Command('Save');
const command2 = new Command('Load');

manager.execute(command1);
manager.execute(command2);
```

30. Implement a Basic Data Fetching with Retry Logic

Question: Write a function that retries a data fetch on failure.
Solution:

```javascript
async function fetchWithRetry(url, options = {}, retries = 3) {
    for (let i = 0; i < retries; i++) {
        try {
            const response = await fetch(url, options);
            if (!response.ok) throw new Error('Network response was not ok');
            return await response.json();
        } catch (error) {
            console.error(`Attempt ${i + 1} failed: ${error.message}`);
            if (i === retries - 1) throw error;
        }
    }
}

// Usage
fetchWithRetry('https://api.example.com/data')
    .then(data => console.log(data))
    .catch(error => console.error('Final error:', error));
```