Advanced JavaScript Concepts Questions

OOP and Prototypes

Question 1: Create a Class with Inheritance

Create a `Shape` class and a `Circle` subclass that calculates the area of the circle.
Solution:

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

    getName() {
        return this.name;
    }
}

class Circle extends Shape {
    constructor(radius) {
        super('Circle');
        this.radius = radius;
    }

    area() {
        return Math.PI * this.radius ** 2;
    }
}

const circle = new Circle(5);
console.log(circle.getName()); // "Circle"
console.log(circle.area());     // 78.53981633974483
```

Question 2: Prototypal Inheritance

Implement a function-based prototype inheritance where `Dog` inherits from `Animal`.
Solution:

```javascript
function Animal(name) {
    this.name = name;
}

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

function Dog(name) {
    Animal.call(this, name);
}

Dog.prototype = Object.create(Animal.prototype);
Dog.prototype.constructor = Dog;

Dog.prototype.speak = function() {
    console.log(`${this.name} barks.`);
};

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

Question 3: Static Methods in Classes

Create a class `MathUtils` with a static method to calculate the factorial of a number.
Solution:

```javascript
class MathUtils {
    static factorial(n) {
        if (n < 0) return undefined; // Factorial not defined for negative numbers
        return n === 0 ? 1 : n * MathUtils.factorial(n - 1);
    }
}

console.log(MathUtils.factorial(5)); // 120
```

Question 4: Method Chaining

Create a class `Calculator` that supports method chaining for basic arithmetic operations.
Solution:

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

    add(num) {
        this.value += num;
        return this; // Return the current instance for chaining
    }

    subtract(num) {
        this.value -= num;
        return this; // Return the current instance for chaining
```

```
    }

    getValue() {
        return this.value;
    }
}

const calc = new Calculator();
const result = calc.add(5).subtract(2).getValue();
console.log(result); // 3
```

Question 5: Object Composition with Classes

Design a `Person` class and a `Job` class, then compose them into a `Worker` class.
Solution:

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

class Job {
    constructor(title) {
        this.title = title;
    }
}

class Worker {
    constructor(name, title) {
        this.person = new Person(name);
        this.job = new Job(title);
    }

    getInfo() {
        return `${this.person.name} is a ${this.job.title}.`;
    }
}

const worker = new Worker('Alice', 'Developer');
console.log(worker.getInfo()); // "Alice is a Developer."
```

Batch 2: `call`, `apply`, `bind`

Question 6: Using `call` to Change Context

Demonstrate how to use `call` to borrow methods from another object.
Solution:

```javascript
const person = {
    firstName: 'John',
    lastName: 'Doe'
};

function fullName() {
    return `${this.firstName} ${this.lastName}`;
```

```
}

console.log(fullName.call(person)); // "John Doe"
```

Question 7: Using `apply` with an Array

Use `apply` to invoke a function with an array of arguments.
Solution:

```javascript
function sum(x, y) {
    return x + y;
}

const numbers = [5, 10];
console.log(sum.apply(null, numbers)); // 15
```

Question 8: Using `bind` to Create a New Function

Use `bind` to create a new function that has a specific context.
Solution:

```javascript
const user = {
    name: 'Jane',
    greet: function() {
        console.log(`Hello, my name is ${this.name}.`);
    }
};

const greetUser = user.greet.bind(user);
greetUser(); // "Hello, my name is Jane."
```

Question 9: Partial Application with `bind`

Implement a function that uses `bind` for partial application of arguments.
Solution:

```javascript
function multiply(a, b) {
    return a * b;
}

const double = multiply.bind(null, 2); // Partial application of the first argument
console.log(double(5)); // 10
```

Question 10: Borrowing Methods

Create an example of borrowing a method using `call` to format an object.

Solution:

```javascript
const user1 = {
    name: 'Alice',
    age: 25,
};

const user2 = {
    name: 'Bob',
    age: 30,
};

function displayInfo() {
    console.log(`${this.name} is ${this.age} years old.`);
}

displayInfo.call(user1); // "Alice is 25 years old."
displayInfo.call(user2); // "Bob is 30 years old."
```

Question 11: Basic Generator Function

Create a basic generator that yields numbers from 1 to 5.

Solution:

```javascript
function* numberGenerator() {
    for (let i = 1; i <= 5; i++) {
        yield i;
    }
}

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

Question 12: Fibonacci Sequence with Generators

Implement a generator function that produces the Fibonacci sequence.

Solution:

```javascript
function* fibonacci() {
    let a = 0, b = 1;
    while (true) {
        yield a;
        [a, b] = [b, a + b];
    }
}

const fib = fibonacci();
console.log(fib.next().value); // 0
```

```
console.log(fib.next().value); // 1
console.log(fib.next().value); // 1
console.log(fib.next().value); // 2
```

Question 13: Infinite Sequence Generator

Create a generator that produces an infinite sequence of natural numbers.
Solution:

```javascript
function* naturalNumbers() {
    let num = 1;
    while (true) {
        yield num++;
    }
}

const natural = naturalNumbers();
console.log(natural.next().value); // 1
console.log(natural.next().value); // 2
```

Question 14: Generator with `return`

Use a generator function that can exit early with a return statement.
Solution:

```javascript
function* limitedGenerator() {
    yield 1;
    yield 2;
    return 'No more values';
    yield 3; // This will never execute
}

const gen = limitedGenerator();
console.log(gen.next()); // { value: 1, done: false }
console.log(gen.next()); // { value: 2, done: false }
console.log(gen.next()); // { value: 'No more values', done: true }
```

Question 15: Yielding Promises

Create a generator that yields promises and handles them using `async/await`.
Solution:

```javascript
function* asyncGenerator() {
    const data1 = yield new Promise(resolve => setTimeout(() => resolve('First value'),
1000));
    const data2 = yield new Promise(resolve => setTimeout(() => resolve('Second value'),
1000));
    return `${data1} and ${data2}`;
}
```

```
async function handleAsyncGenerator(gen) {
    const iterator = gen();
    const res1 = await iterator.next().value;
    const res2 = await iterator.next(res1).value;
    const finalResult = iterator.next(res2).value;
    console.log(finalResult); // "First value and Second value"
}

handleAsyncGenerator(asyncGenerator);
```

Question 16: Class Method Overriding

Create a class `Animal` and subclass `Dog`, overriding a method in the subclass.
Solution:

javascript

```
class Animal {
    speak() {
        return 'Animal makes a noise';
    }
}

class Dog extends Animal {
    speak() {
        return 'Dog barks';
    }
}

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

Question 17: Using `Object.create` for Inheritance

Use `Object.create` to create a prototype-based inheritance structure.
Solution:

javascript

```
const animal = {
    speak() {
        return 'Animal makes a noise';
    }
};

const dog = Object.create(animal);
dog.speak = function() {
    return 'Dog barks';
};

console.log(dog.speak()); // "Dog barks"
console.log(animal.speak()); // "Animal makes a noise"
```

`

Question 19: Composition vs. Inheritance

Demonstrate the difference between composition and inheritance using a `**Vehicle**` class and a `**Car**` class.
Solution:

```javascript
class Vehicle {
    constructor(make) {
        this.make = make;
    }

    start() {
        return `${this.make} is starting.`;
    }
}

class Car {
    constructor(make, model) {
        this.vehicle = new Vehicle(make);
        this.model = model;
    }

    start() {
        return `${this.vehicle.start()} It's a ${this.model}.`;
    }
}

const car = new Car('Toyota', 'Corolla');
console.log(car.start()); // "Toyota is starting. It's a Corolla."
```

Question 20: Mixins for Shared Behavior

Implement a mixin to share behavior between different classes.
Solution:

```javascript
const CanFly = {
    fly() {
        console.log(`${this.name} can fly!`);
```

```
    }
};

class Bird {
    constructor(name) {
        this.name = name;
    }
}

Object.assign(Bird.prototype, CanFly);

const parrot = new Bird('Parrot');
parrot.fly(); // "Parrot can fly!"
```

Question 21: Using `Object.setPrototypeOf`

Demonstrate how to change an object's prototype using **`Object.setPrototypeOf`**.
Solution:

```javascript
const animal = {
    speak() {
        return 'Animal noise';
    }
};

const dog = {
    bark() {
        return 'Woof!';
    }
};

Object.setPrototypeOf(dog, animal);
console.log(dog.speak()); // "Animal noise"
console.log(dog.bark());  // "Woof!"
```

Question 22: Encapsulation with Closures

Create a counter function that encapsulates a private variable.
Solution:

```javascript
function createCounter() {
    let count = 0;

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

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

Question 23: Method Chaining with Object Prototypes

Create an object with methods that allow for chaining.
Solution:

```javascript
const chainable = {
    value: 0,

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

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

    getValue() {
        return this.value;
    }
};

const result = chainable.add(5).subtract(2).getValue();
console.log(result); // 3
```

Question 24: `apply` with Array-Like Objects

Use `apply` to convert an array-like object to an array.
Solution:

```javascript
function logArguments() {
    console.log(Array.from(arguments)); // Convert arguments to an array
}

const arrayLike = {
    0: 'Hello',
    1: 'World',
    length: 2
};

logArguments.apply(null, Array.from(arrayLike)); // ["Hello", "World"]
```

Question 25: Custom `bind` Implementation

Implement your own version of `bind`.
Solution:

```javascript
Function.prototype.myBind = function(context, ...args) {
    const fn = this;
    return function(...newArgs) {
        return fn.apply(context, args.concat(newArgs));
    };
};

const user = {
    name: 'Mike',
    greet: function() {
        console.log(`Hello, my name is ${this.name}.`);
    }
};

const greetMike = user.greet.myBind(user);
greetMike(); // "Hello, my name is Mike."
```

Question 26: Partial Application with Custom Function

Create a function that allows for partial application of multiple arguments.
Solution:

```javascript
function partial(fn, ...presetArgs) {
    return function(...laterArgs) {
        return fn(...presetArgs, ...laterArgs);
    };
}

function multiply(x, y) {
    return x * y;
}

const double = partial(multiply, 2);
console.log(double(5)); // 10
```

Question 27: Generator for Lazy Evaluation

Implement a generator that produces squares of numbers lazily.
Solution:

```javascript
function* squareGenerator() {
    let num = 1;
    while (true) {
        yield num * num;
        num++;
    }
}

const squares = squareGenerator();
console.log(squares.next().value); // 1
```

```
console.log(squares.next().value); // 4
console.log(squares.next().value); // 9
```

Question 28: Using Generators for Asynchronous Flow Control

Use a generator to control the flow of asynchronous code.
Solution:

```javascript
function* asyncFlow() {
    const data1 = yield fetch('https://api.example.com/data1').then(res => res.json());
    const data2 = yield fetch(`https://api.example.com/data2/${data1.id}`).then(res =>
res.json());
    return data2;
}

async function handleAsyncFlow(gen) {
    const iterator = gen();
    const res1 = await iterator.next().value;
    const res2 = await iterator.next(res1).value;
    console.log(res2);
}

handleAsyncFlow(asyncFlow);
```

Question 29: Using Generators to Implement Iterators

Create an iterator using a generator for a custom data structure.
Solution:

```javascript
class CustomArray {
    constructor(...elements) {
        this.elements = elements;
    }

    *[Symbol.iterator]() {
        for (const element of this.elements) {
            yield element;
        }
    }
}

const arr = new CustomArray(1, 2, 3, 4);
for (const num of arr) {
    console.log(num); // 1, 2, 3, 4
}
```

Question 30: Combining Generators with Promises

Create a generator that yields promises and resolves them sequentially.
Solution:

javascript

```javascript
function* fetchData() {
    const res1 = yield fetch('https://api.example.com/data1').then(res => res.json());
    console.log('Data 1:', res1);

    const res2 = yield fetch('https://api.example.com/data2').then(res => res.json());
    console.log('Data 2:', res2);
}

async function handleFetchData(gen) {
    const iterator = gen();

    let result = iterator.next();
    while (!result.done) {
        const promiseResult = await result.value;
        result = iterator.next(promiseResult);
    }
}

handleFetchData(fetchData);
```

`