CODE EXERCISE

🌈 Elevate Your JavaScript Skills with Design Patterns! 🚀

CODING EXERCISES TEST YOUR SKILLS

🌈 Elevate Your JavaScript Skills with Design Patterns! 🚀

Delve into the world of JavaScript Design Patterns with our meticulously crafted series of 10 coding exercises. Each exercise is designed to provide a deep understanding of how to apply these patterns to solve common design problems in your applications.

From creating single instances with the Singleton Pattern to managing complex operations with the Command Pattern, these exercises cover a wide range of scenarios that every JavaScript developer should master.

#JavaScript #DesignPatterns #WebDevelopment #CodingExercises #SoftwareDesign #Programming #DeveloperTools

Dive into these exercises, and share your progress and insights! Let’s foster a learning community where we explore the elegance and efficiency of design patterns together. 🎨👨‍💻

Design patterns in JavaScript are reusable solutions to commonly occurring problems in software design. They can help improve the efficiency of a developer's code and make it more manageable and easier to understand. Below are 10 coding exercises that cover various JavaScript design patterns, complete with explanations and code samples.

Exercise 1: The Singleton Pattern

Objective: Implement a Singleton class that can only be instantiated once.

```
let instance;
class Singleton {
  constructor() {
    if (!instance) {
      instance = this;
```

```
    }
    return instance;
  }
}
// Test the Singleton
const obj1 = new Singleton();
const obj2 = new Singleton();
console.log(obj1 === obj2); // Expected output: true
```

Explanation: The Singleton pattern restricts a class to a single instance. If an instance exists, it returns that; otherwise, it creates one. This pattern is useful when exactly one object is needed to coordinate actions across the system.

Exercise 2: The Module Pattern

Objective: Create a Module pattern to encapsulate private variables and functions.

```
const CalculatorModule = (function() {
  let _data = 0; // private variable
  function add(input) {
    _data += input;
  }
  function getData() {
    return _data;
  }
  return {
    add: add,
```

```
    getData: getData
  };
})();
CalculatorModule.add(5);
console.log(CalculatorModule.getData()); // Expected output: 5
console.log(CalculatorModule._data); // Expected output: undefined
```

Explanation: The Module pattern allows for private variables and functions to be encapsulated within a function scope, exposed only through a returned object. It's a great way to protect variables and methods from being accessed globally.

Exercise 3: The Factory Pattern

Objective: Implement a Factory pattern to create objects without specifying the exact class of object that will be created.

```
class ProductA {
  constructor() {
    this.type = 'Product A';
  }
}
class ProductB {
  constructor() {
    this.type = 'Product B';
  }
}
class ProductFactory {
```

```
  createProduct(type) {
    if (type === 'A') return new ProductA();
    if (type === 'B') return new ProductB();
  }
}
const factory = new ProductFactory();
const productA = factory.createProduct('A');
console.log(productA.type); // Expected output: Product A
```

Explanation: The Factory pattern is useful for creating a class that helps in creating objects without having to specify the exact class of the object that will be created.

Exercise 4: The Observer Pattern

Objective: Implement an Observer pattern to allow objects to subscribe and unsubscribe to events.

```
class Subject {
  constructor() {
    this.observers = [];
  }
  subscribe(observer) {
    this.observers.push(observer);
  }
  unsubscribe(observer) {
    this.observers = this.observers.filter(obs => obs !== observer);
  }
```

```
  notify(data) {
    this.observers.forEach(observer => observer.update(data));
  }
}
class Observer {
  update(data) {
    console.log(`Received data: ${data}`);
  }
}
const subject = new Subject();
const observer = new Observer();
subject.subscribe(observer);
subject.notify('Hello World'); // Expected output: Received data: Hello World
subject.unsubscribe(observer);
subject.notify('Hello again'); // No output expected
```

Explanation: The Observer pattern allows an object, known as the subject, to maintain a list of observers and notify them of state changes, typically by calling one of their methods.

Exercise 5: The Prototype Pattern

Objective: Use the Prototype pattern to clone objects without going through their constructor.

```
const carPrototype = {
  drive() {
```

```
    console.log(`This car is a ${this.make} and it is moving.`);
  }
};
function Car(make) {
  this.make = make;
}
Car.prototype = carPrototype;
const car1 = new Car('Toyota');
car1.drive(); // Expected output: This car is a Toyota and it is moving.
```

Explanation: The Prototype pattern is used to create objects based on a template of an existing object through cloning. It allows for the addition of new properties and methods at runtime.

Exercise 6: The Command Pattern

Objective: Implement the Command pattern to encapsulate a request as an object, thereby allowing for parameterization of clients with queues, requests, and operations.

```
class Light {
  turnOn() { console.log("Light turned on"); }
  turnOff() { console.log("Light turned off"); }
}
class Command {
  constructor(subject) {
    this.subject = subject;
```

```
  }
  execute() {}
}
class TurnOnCommand extends Command {
  execute() {
    this.subject.turnOn();
  }
}
class TurnOffCommand extends Command {
  execute() {
    this.subject.turnOff();
  }
}
const light = new Light();
const turnOnCommand = new TurnOnCommand(light);
const turnOffCommand = new TurnOffCommand(light);
turnOnCommand.execute(); // Expected output: Light turned on
turnOffCommand.execute(); // Expected output: Light turned off
```

Explanation: The Command pattern encapsulates actions or operations as objects, allowing clients to parameterize other objects with different requests, queue operations, or support undoable operations.

Exercise 7: The Strategy Pattern

Objective: Use the Strategy pattern to define a family of algorithms, encapsulate each one, and make them interchangeable.

```
class Strategy {
  execute() { return; }
}
class OperationAdd extends Strategy {
  execute(a, b) { return a + b; }
}
class OperationSubtract extends Strategy {
  execute(a, b) { return a - b; }
}
class Context {
  constructor(strategy) {
    this.strategy = strategy;
  }
  executeStrategy(a, b) {
    return this.strategy.execute(a, b);
  }
}
const contextAdd = new Context(new OperationAdd());
console.log(contextAdd.executeStrategy(3, 1)); // Expected output: 4
const contextSubtract = new Context(new OperationSubtract());
console.log(contextSubtract.executeStrategy(3, 1)); // Expected output: 2
```

Explanation: The Strategy pattern is used to create a set of algorithms that are interchangeable. It allows for the algorithm to vary independently from clients that use it, enabling the client to choose the most suitable algorithm at runtime.

Exercise 8: The Decorator Pattern

Objective: Apply the Decorator pattern to extend the functionality of objects by wrapping them.

```
class Coffee {
  cost() {
    return 5;
  }
}
function withMilk(coffee) {
  const cost = coffee.cost();
  coffee.cost = function() {
    return cost + 1;
  };
}
function withSugar(coffee) {
  const cost = coffee.cost();
  coffee.cost = function() {
    return cost + 0.5;
  };
}
```

```
const myCoffee = new Coffee();
withMilk(myCoffee);
withSugar(myCoffee);
console.log(myCoffee.cost()); // Expected output: 6.5
```

Explanation: The Decorator pattern is used to add new functionality to objects by placing these objects inside special wrapper objects that contain the new functionality.

Exercise 9: The Chain of Responsibility Pattern

Objective: Implement the Chain of Responsibility pattern to pass requests along a chain of handlers.

```
class Handler {
  constructor(successor) {
    this.successor = successor;
  }
  handle(request) {
    if (this.successor) {
      this.successor.handle(request);
    }
  }
}
class ConcreteHandler1 extends Handler {
  handle(request) {
    if (request === 'R1') {
```

```
      console.log('ConcreteHandler1 handled the request');
    } else {
      super.handle(request);
    }
  }
}
class ConcreteHandler2 extends Handler {
  handle(request) {
    if (request === 'R2') {
      console.log('ConcreteHandler2 handled the request');
    } else {
      super.handle(request);
    }
  }
}
const handler1 = new ConcreteHandler1(new ConcreteHandler2());
handler1.handle('R2'); // Expected output: ConcreteHandler2 handled the request
```

Explanation: The Chain of Responsibility pattern allows for the decoupling of the sender of a request from its receivers by giving multiple objects a chance to handle the request.

Exercise 10: The State Pattern

Objective: Utilize the State pattern to alter the behavior of an object when its internal state changes.

```
class State {
  doAction(context) {}
}
class StartState extends State {
  doAction(context) {
    console.log('Player is in start state');
    context.setState(this);
  }
}
class StopState extends State {
  doAction(context) {
    console.log('Player is in stop state');
    context.setState(this);
  }
}
class Context {
  constructor() {
    this.state = null;
  }
  setState(state) {
    this.state = state;
  }
  getState() {
    return this.state;
```

```
  }
}
const context = new Context();
const startState = new StartState();
startState.doAction(context);
console.log(context.getState() instanceof StartState); // Expected output: true
const stopState = new StopState();
stopState.doAction(context);
console.log(context.getState() instanceof StopState); // Expected output: true
```

Explanation: The State pattern is used when the behavior of an object should change according to its internal state. By changing the state, we can make it appear as if the object has changed its class.