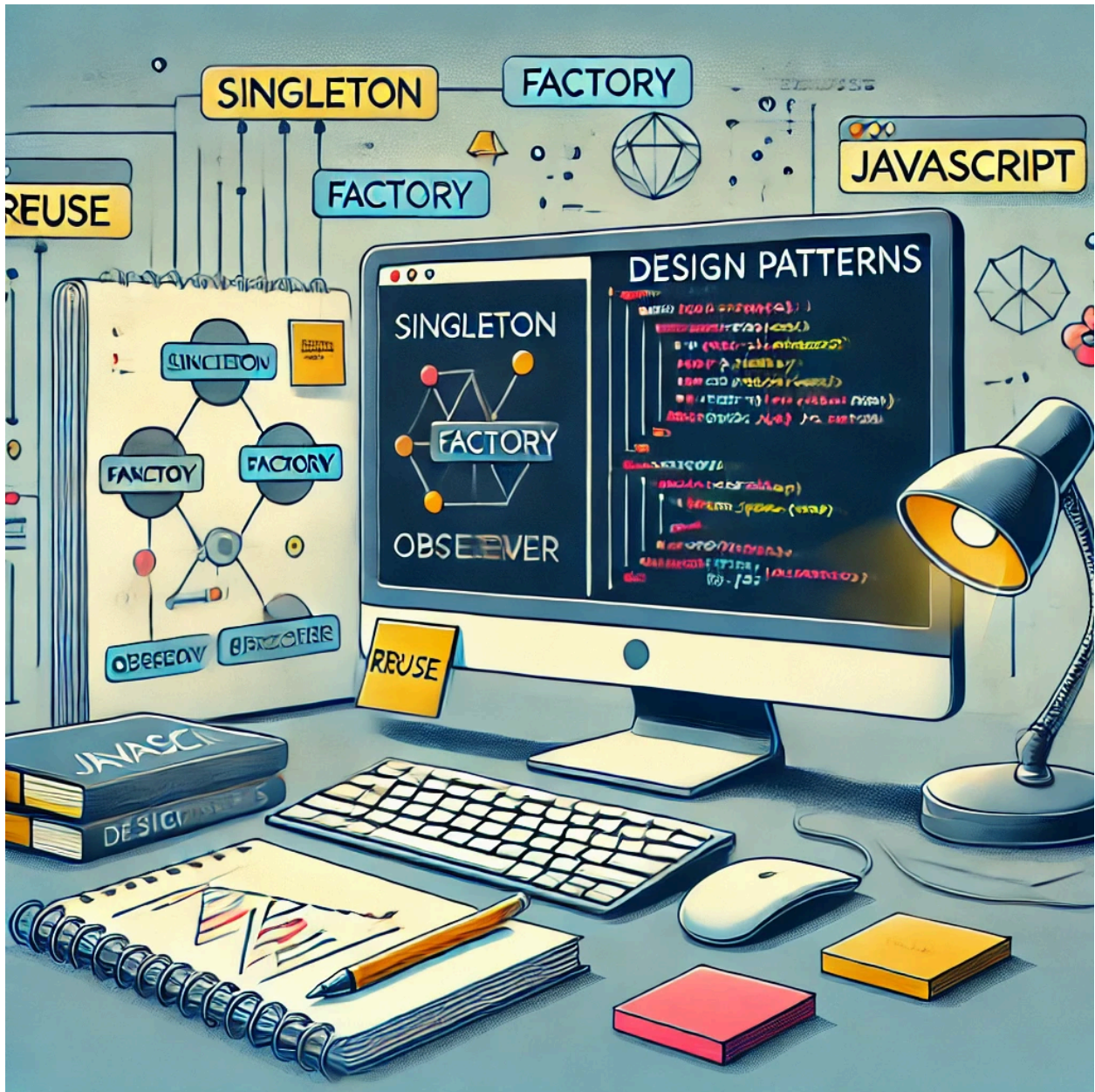


JavaScript Design Patterns: Comprehensive Guide



JavaScript Design Patterns: Comprehensive Guide	1
What Are Design Patterns?	2
Categories of Design Patterns	2
1. Singleton Pattern	2
2. Factory Pattern	3
3. Module Pattern	4

Learn more HTML, CSS, JavaScript Web Development at <https://basescripts.com/> Laurence Svekis

4. Observer Pattern	4
5. Strategy Pattern	5
Exercises	6
Exercise 1: Create a Singleton Logger	6
Exercise 2: Implement a Module	6
Exercise 3: Build an Observer System	6
Exercise 4: Payment Strategy	6
Multiple-Choice Questions	6
Question 1:	6
Question 2:	7
Question 3:	7
Best Practices for Using Design Patterns	7

Design patterns are reusable solutions to common problems in software design. In JavaScript, these patterns improve code readability, maintainability, and scalability. This guide covers some of the most popular JavaScript design patterns, complete with code examples, explanations, multiple-choice questions, and exercises.

What Are Design Patterns?

- **Definition:** Standardized solutions to recurring design problems in programming.
- **Benefits:**
 1. Simplify code organization.
 2. Enhance maintainability.
 3. Provide a shared vocabulary for developers.

Categories of Design Patterns

1. **Creational Patterns:** Focus on object creation.
 - Examples: Singleton, Factory
2. **Structural Patterns:** Organize relationships between objects.
 - Examples: Module, Decorator
3. **Behavioral Patterns:** Manage communication between objects.
 - Examples: Observer, Strategy

1. Singleton Pattern

Ensures a class has only one instance and provides a global point of access to it.

Example:

```
const Singleton = (function () {
```

Learn more HTML, CSS, JavaScript Web Development at <https://basescripts.com/> Laurence Svekis

```

let instance;
function createInstance() {
  return { name: "Singleton Instance" };
}
return {
  getInstance() {
    if (!instance) {
      instance = createInstance();
    }
    return instance;
  },
};
})();
const instance1 = Singleton.getInstance();
const instance2 = Singleton.getInstance();
console.log(instance1 === instance2); // Output: true

```

Explanation:

- **Use Case:** Managing a single configuration or logging instance across an application.

2. Factory Pattern

Creates objects without specifying the exact class of object to create.

Example:

```

function CarFactory() {
  this.createCar = function (type) {
    if (type === "SUV") {
      return { type: "SUV", wheels: 4 };
    } else if (type === "Truck") {
      return { type: "Truck", wheels: 6 };
    }
  };
}
const factory = new CarFactory();
const suv = factory.createCar("SUV");
const truck = factory.createCar("Truck");

```

Learn more HTML, CSS, JavaScript Web Development at <https://basescripts.com/> Laurence Svekis

```
console.log(suv, truck);
```

Explanation:

- **Use Case:** Simplify object creation logic based on input parameters.

3. Module Pattern

Encapsulates code into a single object to avoid polluting the global namespace.

Example:

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

Explanation:

- **Use Case:** Create reusable and isolated code components.

4. Observer Pattern

Allows objects to subscribe to events and get notified when changes occur.

Example:

```
class Observable {
  constructor() {
    this.observers = [];
  }
  subscribe(observer) {
    this.observers.push(observer);
  }
}
```

Learn more HTML, CSS, JavaScript Web Development at <https://basescripts.com/> Laurence Svekis

```

    }
    unsubscribe(observer) {
      this.observers = this.observers.filter((obs) => obs !== observer);
    }
    notify(data) {
      this.observers.forEach((observer) => observer(data));
    }
  }
}
const observable = new Observable();
const observer1 = (data) => console.log("Observer 1:", data);
const observer2 = (data) => console.log("Observer 2:", data);
observable.subscribe(observer1);
observable.subscribe(observer2);
observable.notify("Data updated"); // Output: Observer 1: Data
updated; Observer 2: Data updated

```

Explanation:

- **Use Case:** Event systems, like React's state updates or DOM events.

5. Strategy Pattern

Defines a family of algorithms and makes them interchangeable.

Example:

```

class PaymentProcessor {
  constructor(strategy) {
    this.strategy = strategy;
  }
  process(amount) {
    return this.strategy.pay(amount);
  }
}
class CreditCard {
  pay(amount) {
    return `Paid $$${amount} using Credit Card`;
  }
}

```

Learn more HTML, CSS, JavaScript Web Development at <https://basescripts.com/> Laurence Svekis

```
class PayPal {
  pay(amount) {
    return `Paid $$${amount} using PayPal`;
  }
}

const creditCardPayment = new PaymentProcessor(new CreditCard());
console.log(creditCardPayment.process(100)); // Output: Paid $100
using Credit Card
const paypalPayment = new PaymentProcessor(new PayPal());
console.log(paypalPayment.process(200)); // Output: Paid $200 using
PayPal
```

Explanation:

- **Use Case:** Payment gateways, sorting algorithms, or other interchangeable processes.

Exercises

Exercise 1: Create a Singleton Logger

Write a logger class that ensures only one instance exists and can log messages to the console.

Exercise 2: Implement a Module

Create a `MathModule` with private variables and public methods for addition and multiplication.

Exercise 3: Build an Observer System

Implement an observer pattern for a stock price tracker where multiple observers are notified when the price updates.

Exercise 4: Payment Strategy

Extend the Strategy Pattern example to add a `CryptoPayment` method.

Multiple-Choice Questions

Question 1:

Which pattern ensures only one instance of a class exists?

1. Factory Pattern
2. Singleton Pattern

Learn more HTML, CSS, JavaScript Web Development at <https://basescripts.com/> Laurence Svekis

3. Observer Pattern
4. Strategy Pattern

Answer: 2. Singleton Pattern

Question 2:

What does the Module Pattern primarily help with?

1. Managing state in real-time applications.
2. Encapsulating code and avoiding global namespace pollution.
3. Dynamically creating objects based on input.
4. Allowing multiple observers to listen to events.

Answer: 2. Encapsulating code and avoiding global namespace pollution.

Question 3:

Which pattern is ideal for implementing payment gateways with different providers?

1. Singleton Pattern
2. Factory Pattern
3. Strategy Pattern
4. Module Pattern

Answer: 3. Strategy Pattern

Best Practices for Using Design Patterns

1. **Understand the Problem:** Choose the pattern that best fits the use case.
2. **Keep It Simple:** Avoid overcomplicating code with unnecessary patterns.
3. **Modularize Code:** Use patterns like Module and Factory for reusable and scalable components.
4. **Document Patterns:** Ensure all team members understand the chosen patterns.