Design patterns (Module, Singleton, Factory, Observer)

Let's explore **four common JavaScript design patterns**: **Module**, **Singleton**, **Factory**, and **Observer**, with detailed explanations and code examples.

---

1. Module Pattern

Purpose:

Encapsulate private variables and expose only what's necessary via a public API.

Use Case:

Data privacy, code organization, and avoiding polluting the global namespace.

Syntax Example:

```
const CounterModule = (function () {
  let count = 0; // private variable

  return {
    increment: function () {
      count++;
      console.log(count);
    },
    reset: function () {
      count = 0;
      console.log("Reset:", count);
    }
  };
})();

CounterModule.increment(); // 1
CounterModule.increment(); // 2
CounterModule.reset();     // Reset: 0
```

✅ **Key Benefit:** Private scope using closures.

---

2. Singleton Pattern

Purpose:

Ensure only one instance of a class/object exists.

Use Case:

Application settings, configuration managers, or caching.

Syntax Example:

```
const Singleton = (function () {
  let instance;

  function createInstance() {
    return { name: "SingletonInstance", time: Date.now() };
  }

  return {
    getInstance: function () {
      if (!instance) {
        instance = createInstance();
      }
      return instance;
    }
  };
})();

const obj1 = Singleton.getInstance();
const obj2 = Singleton.getInstance();
console.log(obj1 === obj2); // true
```

✅ **Key Benefit:** Global point of access to a single instance.

---

3. Factory Pattern

Purpose:

Create objects without specifying the exact class/type.

Use Case:

Creating different types of objects dynamically based on input.

Syntax Example:

```
function Car(type) {
  this.type = type;
  this.drive = function () {
    console.log(`Driving a ${this.type} car`);
  };
}

function Truck(type) {
  this.type = type;
  this.drive = function () {
    console.log(`Driving a ${this.type} truck`);
  };
}

function VehicleFactory() {
  this.createVehicle = function (type) {
    if (type === "car") return new Car("Sedan");
    if (type === "truck") return new Truck("Pickup");
  };
}

const factory = new VehicleFactory();
const myCar = factory.createVehicle("car");
myCar.drive(); // Driving a Sedan car
```

✅ **Key Benefit:** Flexibility in object creation logic.

---

4. Observer Pattern

Purpose:

A subject (observable) maintains a list of dependents (observers) and notifies them of changes.

Use Case:

Event handling systems, real-time updates (e.g., chat, stock tickers).

Syntax Example:

```
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(data));
  }
}

// Usage
const newsChannel = new Subject();

function subscriber1(news) {
  console.log("Subscriber 1 received:", news);
}

function subscriber2(news) {
  console.log("Subscriber 2 received:", news);
```

```
}

newsChannel.subscribe(subscriber1);
newsChannel.subscribe(subscriber2);

newsChannel.notify("Breaking News!");
// Both subscribers are notified

newsChannel.unsubscribe(subscriber2);
newsChannel.notify("Second News");
// Only Subscriber 1 is notified
```

✅ **Key Benefit:** Loose coupling between the subject and its observers.

Summary Table

Pattern	Purpose	Best For
Module	Encapsulation using closures	Utilities, private data
Singleton	Single shared instance	Configs, Logging, Database Conn.
Factory	Object creation abstraction	UI elements, API handlers
Observer	One-to-many dependency (event subscription)	Realtime apps, Pub/Sub systems

Let me know if you want real-world use cases or advanced examples of these patterns.