PATRONES DE DISEÑO EN JAVASCRIPT

Objetivo de la Clase:
Los estudiantes aprenderán los conceptos y la implementación de patrones de diseño comunes en JavaScript, como Singleton, Observer, Module, y Factory. Al final de la clase, realizarán retos prácticos para afianzar los conceptos y tendrán una pequeña tarea para reforzar lo aprendido.

Estructura de la clase

- Introducción a los Patrones de Diseño
- Patrón Singleton
- Patrón Observer
- Patrón Module
- Patrón Factory
- Retos en Clase
- Comida

Introducción a los Patrones de Diseño

¿Qué son los Patrones de Diseño?

- Los patrones de diseño son soluciones generales y reutilizables para problemas comunes en el diseño de software.
- Se pueden ver como plantillas para abordar ciertos problemas de diseño de manera estructurada y comprobada.

Importancia de los Patrones de Diseño

- Mejoran la eficiencia en la resolución de problemas recurrentes.
- Facilitan la comunicación entre desarrolladores usando un lenguaje común.
- Ayudan a mantener un código más organizado, modular y mantenible.

Patrón Singleton

¿Qué es el Patrón Singleton?

El patrón Singleton asegura que una clase El Patrón Singleton es un patrón de diseño creacional que asegura que una clase tenga una única instancia y proporciona un punto de acceso global a esa instancia. En otras palabras, el Singleton restringe la instanciación de una clase a un solo objeto.una única instancia y proporciona un punto de acceso global a ella.

¿Cuándo usar el Patrón Singleton?

- Solo debe existir una instancia de una clase, como en el caso de un manejador de configuración o una conexión a una base de datos.
- Necesitas un punto de acceso global para esa instancia, permitiendo que diferentes partes del código accedan y utilicen la misma instancia sin crear copias adicionales.

Patrón Singleton

Ventajas del Patrón Singleton

- Control de Instancias: Garantiza que solo haya una instancia de la clase en toda la aplicación, lo que puede ser útil para manejar recursos compartidos como configuraciones o conexiones a bases de datos.
- Punto de Acceso Global: Ofrece un punto de acceso global a la instancia, lo que facilita la comunicación entre diferentes partes del código que necesitan utilizar el mismo recurso.
- Memoria: Al asegurarse de que solo haya una instancia, el uso de memoria puede ser más eficiente, ya que no se crean múltiples objetos innecesariamente.

Desventajas del Patrón Singleton

- Dificultad en Pruebas: Los Singletons pueden ser difíciles de probar porque su estado persiste a lo largo de la aplicación, lo que puede llevar a efectos secundarios inesperados entre las pruebas.
- Acoplamiento: Puede aumentar el acoplamiento en el código, ya que muchas partes de la aplicación dependen de la misma instancia global.
- Limitación de Flexibilidad: En aplicaciones más grandes o complejas, el uso de Singletons puede limitar la flexibilidad, ya que es difícil cambiar la implementación del Singleton sin afectar a toda la aplicación.

Implementación del Patrón Singleton en JavaScript

```
class Singleton {
    constructor() {
        // Si ya existe una instancia, se retorna la existente
        if (!Singleton.instance) {
            Singleton.instance = this;  // Se guarda la instancia en una propiedad estática
        }

        // Si no existe, se crea una nueva instancia y se guarda
        return Singleton.instance;  // Siempre se retorna la misma instancia
    }
}

// Crear las instancias
const instance1 = new Singleton();
const instance2 = new Singleton();

// Ambas instancias son iguales porque el patrón Singleton asegura que solo haya una instancia
console.log(instance1 === instance2);  // true
```

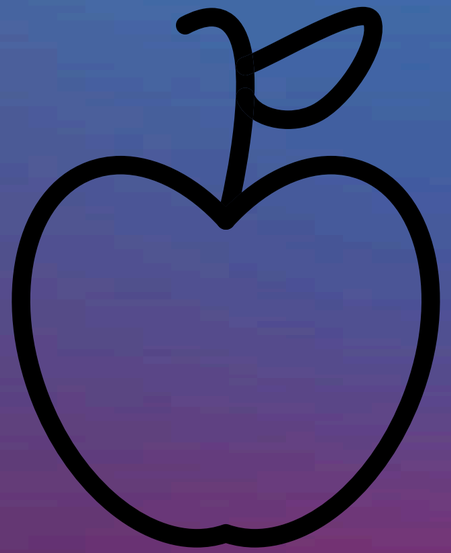
Reto Rápido: Implementación de un Singleton en JavaScript

Descripción del Reto

Implementa un Singleton que maneje la configuración global de una aplicación. Este Singleton debe almacenar un conjunto de configuraciones (como el idioma, el tema de la interfaz, y el modo de operación) y permitir la lectura y actualización de estas configuraciones desde cualquier parte de la aplicación.

Requisitos

1. Crear la clase AppConfig:
   - La clase debe ser un Singleton, es decir, solo puede existir una instancia de AppConfig.
   - Debe tener propiedades para las configuraciones language (idioma), theme (tema de la interfaz), y mode (modo de operación).
2. Métodos:
   - getConfig(): Devuelve el objeto con todas las configuraciones actuales.
   - updateConfig(newConfig): Permite actualizar las configuraciones. newConfig es un objeto que contiene las nuevas configuraciones.
3. Comportamiento:
   - Si intentas crear una segunda instancia de AppConfig, debe devolver la primera instancia creada.
   - Los métodos getConfig() y updateConfig(newConfig) deben reflejar los cambios en la misma instancia compartida



Patrón Observer

¿Qué es el Patrón Observer?

El Patrón Observer es un patrón de diseño comportamental que define una relación de uno a muchos entre objetos, de manera que cuando un objeto cambia de estado, todos sus dependientes (u "observadores") son notificados y actualizados automáticamente. En otras palabras, permite que un objeto ("sujeto" o "subject") notifique a otros objetos ("observadores" o "observers") sobre cambios sin que estos objetos estén acoplados entre sí.

¿Cuándo usar el Patrón Observer?

- Varias partes de tu aplicación necesitan reaccionar a cambios en el estado de un objeto
- Quieres mantener un sistema de notificaciones entre objetos que sea flexible y fácil de extender.
- Necesitas reducir el acoplamiento entre los objetos que generan eventos y los que reaccionan a ellos.

Ejemplo de uso común

Un caso típico del patrón Observer es en interfaces gráficas, donde un cambio en un modelo de datos (como el contenido de un formulario) debe reflejarse automáticamente en la vista (como los campos de entrada) sin que la vista esté directamente acoplada al modelo.

Ventajas del Patrón Observer

- Desacoplamiento: Permite que el sujeto y los observadores estén desacoplados entre sí, facilitando la extensibilidad y el mantenimiento.
- Escalabilidad: Se pueden añadir más observadores sin cambiar el código del sujeto. Esto hace que el sistema sea escalable y fácil de modificar.
- Flexibilidad: Los observadores pueden registrarse y eliminarse en tiempo de ejecución, lo que permite una gran flexibilidad en la aplicación.

Desventajas del Patrón Observer

- Complejidad: Puede agregar complejidad a la aplicación, especialmente si se tienen muchos observadores, lo que podría dificultar el seguimiento del flujo de eventos.
- Eficiencia: Si hay demasiados observadores, o si los observadores realizan operaciones costosas, el rendimiento puede verse afectado negativamente.
- Potenciales fallas de seguridad: Dado que los observadores pueden registrarse dinámicamente, podría ser más difícil controlar qué observadores están escuchando los eventos, lo que puede llevar a problemas de seguridad si no se maneja correctamente.

Implementación del Patrón Observer en JavaScript

```
// Clase Subject (Sujeto)
class Subject {
    constructor() {
        this.observers = [];  // Array para almacenar los observadores
    }

    // Añadir un observador
    addObserver(observer) {
        this.observers.push(observer);
    }

    // Eliminar un observador
    removeObserver(observer) {
        this.observers = this.observers.filter(obs => obs !== observer);
    }

    // Notificar a todos los observadores sobre un cambio
    notify(data) {
        this.observers.forEach(observer => observer.update(data));
    }
}

// Clase Observer (Observador)
class Observer {
    update(data) {
        console.log('Observador recibió la notificación con datos:', data);
    }
}
```

```
// Ejemplo de uso
const subject = new Subject();

const observer1 = new Observer();
const observer2 = new Observer();

// Añadir observadores
subject.addObserver(observer1);
subject.addObserver(observer2);

// Notificar a todos los observadores
subject.notify('Un evento importante ocurrió');

// Eliminar un observador y notificar nuevamente
subject.removeObserver(observer1);
subject.notify('Otro evento ocurrió');
```

Reto de Programación: Implementación del Patrón Observer en JavaScript

Descripción del Reto

Imagina que estás desarrollando una aplicación de noticias en la que diferentes usuarios pueden suscribirse a ciertos canales de noticias (como deportes, tecnología, política, etc.). Cuando se publica una nueva noticia en un canal, todos los usuarios suscritos a ese canal deben ser notificados. Tu tarea es implementar este sistema utilizando el Patrón Observer

Reto de Programación: Implementación del Patrón Observer en JavaScript

Requisitos

1. Crear la clase NewsChannel:
   - Esta clase actuará como el sujeto (Subject).
   - Debe permitir que los usuarios se suscriban (subscribe), se desuscriban (unsubscribe), y recibir notificaciones cuando se publica una nueva noticia (publishNews).
2. Crear la clase User:
   - Esta clase representará a los observadores (Observers).
   - Debe tener un método update que reciba las noticias publicadas y muestre un mensaje con la noticia recibida.
3. Métodos del NewsChannel:
   - subscribe(user): Añade un usuario a la lista de suscriptores.
   - unsubscribe(user): Elimina un usuario de la lista de suscriptores.
   - publishNews(news): Notifica a todos los usuarios suscritos sobre la nueva noticia.
4. Comportamiento Esperado:
   - Cuando se publique una nueva noticia, todos los usuarios suscritos al canal deben recibir la noticia a través del método update.
   - Los usuarios pueden suscribirse o desuscribirse en cualquier momento.

Patrón Module

¿Qué es el Patrón Module?

El Patrón Module es un patrón de diseño que permite organizar y encapsular código en una estructura que imita el concepto de "módulos". En JavaScript, el patrón Module se utiliza para crear "módulos" que encapsulan datos y métodos, controlando qué partes del código son públicas y accesibles desde fuera del módulo y cuáles son privadas.

Características Principales del Patrón Module:

- Encapsulación.El patrón Module permite encapsular código dentro de un "módulo", haciendo que algunas partes del código sean privadas y otras públicas

- API Pública.Solo los métodos y propiedades que se devuelven desde el módulo son accesibles desde fuera, lo que crea una API controlada y bien definida para interactuar con el módulo.

- Prevención de Contaminación del Espacio de Nombres. Al encapsular código en módulos, se evita que las variables y funciones queden expuestas en el espacio de nombres global, lo que reduce la probabilidad de conflictos entre diferentes partes del código.

```
const myModule = (function() {
    // Variables y funciones privadas
    let privateVariable = 'Soy privado';

    function privateMethod() {
        console.log('Este es un método privado');
    }

    // API pública
    return {
        publicMethod: function() {
            console.log('Este es un método público');
            privateMethod();  // Llamada a un método privado
            console.log(privateVariable);  // Acceso a una variable privada
        }
    };
})();

// Uso del módulo
myModule.publicMethod();
// Salida esperada:
// Este es un método público
// Este es un método privado
// Soy privado

// No se puede acceder directamente a las variables y métodos privados
// myModule.privateMethod();  // Error: myModule.privateMethod is not a
function
// console.log(myModule.privateVariable);  // Error: undefined
```

Cómo Implementar el Patrón Module en JavaScript

El patrón Module se puede implementar en JavaScript utilizando funciones autoinvocadas (IIFE, por sus siglas en inglés) que retornan un objeto con los métodos públicos. Aquí hay un ejemplo básico

Mini Reto: Implementación del Patrón Module en JavaScript

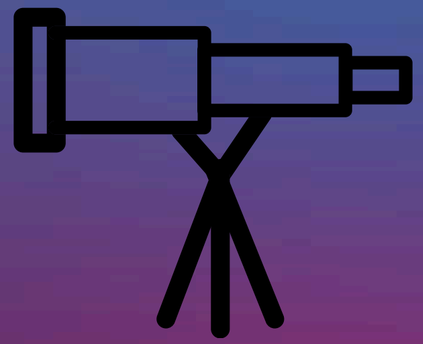
Descripción del Reto

Vas a crear un módulo en JavaScript para gestionar un contador. Este contador deberá poder incrementarse, decrementarse y resetearse, pero la variable que almacena el valor del contador debe ser privada, de modo que solo se pueda modificar a través de los métodos públicos del módulo.

Mini Reto: Implementación del Patrón Module en JavaScript

Requisitos

1. Crear el Módulo counterModule:
   - Debe tener una variable privada que almacene el valor del contador.
   - El valor del contador debe comenzar en 0.
2. Métodos Públicos:
   - increment(): Incrementa el valor del contador en 1.
   - decrement(): Decrementa el valor del contador en 1.
   - reset(): Resetea el valor del contador a 0.
   - getValue(): Retorna el valor actual del contador.
3. Comportamiento Esperado:
   - Los métodos increment y decrement deben modificar el valor del contador sin exponer la variable privada.
   - El método getValue debe permitir ver el valor actual del contador.
   - El método reset debe restaurar el contador a 0.



Patrón Factory en JavaScript

¿Qué es el Patrón Factory?

El Patrón Factory es un patrón de diseño creacional que se utiliza para crear objetos sin tener que especificar la clase exacta del objeto que se creará. En lugar de usar el operador new directamente en el código, el patrón Factory proporciona una interfaz para crear objetos, lo que permite que el proceso de instanciación sea más flexible y dinámico.

Características Principales del Patrón Factory

- Encapsulación de la Creación de Objetos.El proceso de creación de objetos se encapsula en una "fábrica" que decide qué tipo de objeto crear y devuelve una instancia del mismo.
- Desacoplamiento del Código.El código cliente no necesita conocer los detalles de la implementación de los objetos que está utilizando.
- Flexibilidad.Permite añadir nuevas clases de objetos sin modificar el código cliente, lo que facilita la expansión y el mantenimiento del sistema.

```
// Ejemplo básico de una Fábrica (Factory)
function CarFactory(type) {
    let car;

    if (type === 'sedan') {
        car = { type: 'Sedan', doors: 4 };
    } else if (type === 'suv') {
        car = { type: 'SUV', doors: 5 };
    } else if (type === 'convertible') {
        car = { type: 'Convertible', doors: 2 };
    } else {
        throw new Error('Tipo de coche no soportado');
    }

    return car;
}

// Uso de la Fábrica
const sedan = CarFactory('sedan');
console.log(sedan);  // Salida: { type: 'Sedan', doors: 4 }

const suv = CarFactory('suv');
console.log(suv);  // Salida: { type: 'SUV', doors: 5 }

const convertible = CarFactory('convertible');
console.log(convertible);  // Salida: { type: 'Convertible', doors: 2 }
```

Implementación del Patrón Factory en JavaScript

En JavaScript, el patrón Factory se puede implementar utilizando funciones que retornan instancias de objetos. Aquí hay un ejemplo básico

Mini Reto: Implementación del Patrón Module en JavaScript

Descripción del Reto

Vas a crear una fábrica de vehículos que pueda crear diferentes tipos de vehículos (Coche, Motocicleta y Camión). Cada tipo de vehículo tendrá propiedades y métodos específicos, y la fábrica debe ser capaz de devolver una instancia del vehículo solicitado según el tipo.

Mini Reto: Implementación del Patrón Module en JavaScript

Requisitos

1. Crear una Fábrica VehicleFactory:
   - La fábrica debe recibir un tipo de vehículo (car, motorcycle, truck) y devolver una instancia del vehículo correspondiente.
2. Tipos de Vehículos:
   - Coche (Car):
     - Propiedades: type (valor: "Car"), wheels (valor: 4).
     - Método: drive() que imprime "Driving a car!".
   - Motocicleta (Motorcycle):
     - Propiedades: type (valor: "Motorcycle"), wheels (valor: 2).
     - Método: ride() que imprime "Riding a motorcycle!".
   - Camión (Truck):
     - Propiedades: type (valor: "Truck"), wheels (valor: 6).
     - Método: haul() que imprime "Hauling with a truck!".
3. Comportamiento Esperado:
   - La fábrica debe crear y devolver un objeto correspondiente al tipo de vehículo solicitado con las propiedades y métodos adecuados.
   - Si se solicita un tipo de vehículo que no está soportado, la fábrica debe lanzar un error.

Juego del ahorcado

Pista

Estructura de control que permite repetir un bloque de código mientras se cumple una condición

Juego del ahorcado

Pista

Tipo de variable que puede contener más de un valor, almacenándolos en índices numerados

Juego del ahorcado

Pista

Estructura que permite a una función recordar y acceder a su ámbito léxico incluso cuando se ejecuta fuera de ese ámbito

Juego del ahorcado

Pista

Tipo de dato que
puede ser verdadero
o falso

Juego del ahorcado

Pista

Estructura que se usa para almacenar datos en pares clave-valor.

Juego del ahorcado

Pista

Bloque de código reutilizable que puede ser invocado desde cualquier parte del programa.

Juego del ahorcado

Pista

Operador que se utiliza para comparar dos valores, sin tener en cuenta el tipo de datos.

Juego del ahorcado

Pista

Acción de retrasar la ejecución de código hasta que ocurra un evento o condición.

Juego del ahorcado

Pista

Método que permite combinar dos o más arrays en uno solo.

Juego del ahorcado

Pista

Operador que se utiliza para seleccionar elementos dentro del DOM.

Juego del ahorcado

Pista

Nombre que hace referencia a una dirección en memoria que almacena un valor.

Juego del ahorcado

Pista

Declaración que permite ejecutar un bloque de código basándose en una condición.

Juego del ahorcado

Pista

Proceso de llamar una función desde dentro de sí misma.

Recursos gratuitos

Puedes cambiar el color de estos íconos e ilustraciones gratuitos y usarlos en tu diseño de Canva.