TareaPatrones

Carlos Noguez Juárez

May 2024

1 Primer codigo.

```
class Database {
  constructor() {
    if (!Database.instance) {
      Database.instance = this;
    }
    return Database.instance;
  }

  query(sql) {
    console.log("Ejecutando consulta:", sql);
  }
}

const db1 = new Database();
const db2 = new Database();

console.log(db1 === db2); // Output: true
db1.query("SELECT * FROM users");
```

La clase `Database` implementa el patrón Singleton mediante el uso de una variable estática `instance` y una comprobación en el constructor. Cuando se crea una nueva instancia de la clase `Database`, se verifica si ya existe una instancia previa. Si no hay una instancia previa, se crea una nueva y se asigna a la variable `instance`. Si ya existe una instancia previa, se devuelve esa instancia en lugar de crear una nueva.

Aquí está cómo funciona el código:

1. La primera vez que se llama a `new Database()`, se crea una nueva instancia de la clase `Database` y se asigna a la variable estática `Database.instance`.
2. Cuando se llama a `new Database()` por segunda vez (y las siguientes veces), la condición `if (!Database.instance)` será falsa, y en su lugar se devolverá la instancia previamente creada (`Database.instance`).

Esto se demuestra en la última línea del código `console.log(db1 === db2); // true`, donde se muestra que `db1` y `db2` son referencias a la misma instancia de la clase `Database`.

2 Segundo codigo.

```
class Logger {
  constructor() {
    this.logs = [];
  }

  log(message) {
    this.logs.push(message);
    console.log("Log registrado:", message);
  }

  static getInstance() {
    if (!Logger.instance) {
      Logger.instance = new Logger();
    }
    return Logger.instance;
  }
}

const logger1 = Logger.getInstance();
const logger2 = Logger.getInstance();

console.log(logger1 === logger2); // Output: true
logger1.log("Error: No se puede conectar al servidor");
```

Se utilizq el patrón Singleton. Este patrón garantiza que una clase tenga una única instancia y proporcione un punto de acceso global a ella.

En el código, la clase `logger` implementa el patrón Singleton a través del método estático `getInstance()`. Este método verifica si ya existe una instancia de la clase `logger`. Si no existe, se crea una nueva instancia y se asigna a la propiedad estática `logger.instance`. Si ya existe una instancia, se devuelve esa instancia en lugar de crear una nueva.

Aquí está cómo funciona el código:

1. La primera vez que se llama a `logger.getInstance()`, se crea una nueva instancia de la clase `logger` y se asigna a la propiedad estática `logger.instance`.

2. Cuando se llama a `logger.getInstance()` por segunda vez (y las siguientes veces), la condición `if (!logger.instance)` será falsa, y en su lugar se devolverá la instancia previamente creada (`logger.instance`).

Esto se demuestra en la última línea del código `console.log(logger1 === logger2); // true`, donde se muestra que `logger1` y `logger2` son referencias a la misma instancia de la clase `logger`.

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

  greet() {
    console.log("Hola, soy", this.name);
  }
}

class UserFactory {
  createUser(name) {
    return new User(name);
  }
}

const factory = new UserFactory();
const user1 = factory.createUser("Juan");
const user2 = factory.createUser("Maria");

user1.greet(); // Output: Hola, soy Juan
user2.greet(); // Output: Hola, soy Maria
```

3 Tercer codigo

El código proporcionado ilustra la implementación del patrón Factory, un patrón de diseño creacional que centraliza y abstrae el proceso de creación de objetos. En lugar de crear instancias de una clase directamente, se delega esa responsabilidad a una clase separada conocida como "Fábrica". Esta clase actúa como una interfaz que oculta los detalles de creación de objetos.

En el ejemplo, la clase `UserFactory` asume el rol de la Fábrica. Contiene un método estático `create` que se encarga de instanciar objetos de la clase `User`. De esta manera, la lógica de creación de objetos `User` queda encapsulada dentro de `UserFactory`, promoviendo un bajo acoplamiento entre el código que crea los objetos y el código que los utiliza.

El flujo de trabajo se desarrolla de la siguiente manera: primero, se define la clase `User` que representa el objeto a crear, con un constructor que recibe el nombre del usuario y un método `greet` para imprimir un saludo. Luego, se implementa la clase `UserFactory` con el método estático `create`, el cual toma el nombre del usuario como argumento y devuelve una nueva instancia de `User` con ese nombre.

Para crear nuevos usuarios, se instancia `UserFactory` y se invocan llamadas a `factory.create('Juan')` y `factory.create('Pedro')`, creando así dos nuevas instancias de `User` con los nombres proporcionados. Finalmente, se llaman los métodos `greet` en las instancias de usuario creadas para imprimir los saludos correspondientes.