Instancias únicas

¡Comencemos!

¿Qué aprenderás?

- Comprender el Patrón de diseño Singleton para obtener un código limpio al ejecutar.
- Comprender el concepto Synchronized para evitar ejecutar hilos en paralelo.

Introducción

Conoceremos el patrón Singleton de POO, un patrón básico de la programación que logra reducir el uso de memoria si es bien implementado en el software. Es una instancia que se utiliza día a día por múltiples desarrolladores alrededor del mundo, varios/as utilizan este patrón para hacer solicitudes o "request" a través de la web, por ejemplo. Es esencial aprender este concepto para realizar instancias únicas e irrepetibles.

A continuación, veremos qué es Singleton y cómo se utiliza.

¡Vamos con todo!

¿Qué es Singleton?

Singleton es un patrón de diseño de software que se caracteriza porque los objetos del software se rigen por el patrón, solo se instancian una vez y esa instancia es la que se utilizará en toda la aplicación. Te preguntarás de qué sirve aplicar este patrón... pues reduce la cantidad de instancias durante la ejecución de la aplicación y ayuda a tener un código más limpio porque también reduce la cantidad de variables. La ventaja más importante es que si hay varias partes de la aplicación que comparten un mismo recurso, podrán acceder a él desde cualquier parte.

La idea del patrón Singleton es proveer un mecanismo para limitar el número de instancias de una clase. Por lo tanto, el mismo objeto es siempre compartido por distintas partes del código. Es visto como una solución elegante para una variable global, porque los datos son abstraídos por detrás de la interfaz que publica la clase singleton.

¿Cómo se aplica el patrón Singleton?

Para aplicarlo se debe crear una clase que se instancie a sí misma en un contexto `static` y que tenga un constructor privado para que no se pueda acceder a él. Por último, se debe crear un método `static` para que las otras clases puedan acceder a la instancia existente.

Realicemos un ejercicio guiado para ponerlo en práctica:

Ejercicio Guiado: Instituto Educativo

Paso 1: Crear la clase `InstitutoEducativo` y colocar una variable del mismo tipo que la clase pero llamada "instance". Aquí reside el secreto de este patrón, ya que dicha variable es la que se instancia por única vez y se devuelve al cliente.

```
public class InstitutoEducativo {
   private static InstitutoEducativo instance;
   }
```

Paso 2: Privatizar el constructor para que no se pueda hacer new `InstitutoEducativo()` desde otro lugar que no sea dentro de la misma clase.

```
 private InstitutoEducativo() {}
```

Paso 3: Para utilizar la única instancia de la clase, los clientes deberán convocar al método `getInstance()`. Crear la condición `if` que solo será `true` la primera vez.

```
public static InstitutoEducativo getInstance() {
      if (instance == null) {
            instance = new InstitutoEducativo();
      }
      return instance;
   }
```

Realizando los pasos 1, 2 y 3 quedaría de la siguiente manera:

```
public class InstitutoEducativo {
   private static InstitutoEducativo instance;
   private InstitutoEducativo() {}
   public static InstitutoEducativo getInstance() {
      if (instance == null) {
            instance = new InstitutoEducativo();
      }
      return instance;
   }
}
```

Paso 4: Para llamar al instituto, obtener la instancia del instituto en el Main.

```
public class Main{

public static void main(String[] args){
InstitutoEducativo instituto = InstitutoEducativo.getInstance();
 }
}
```

Otro ejemplo de Singleton

Ahora veamos otro ejemplo donde podemos ocupar Singleton. Para ello crearemos la clase `Configurador()` en un proyecto aparte y definiremos los siguientes métodos e instancias:

```
public class Configurador{

 //Variable encapsulada y estática donde se almacenará la instancia.

  private static Configurador config;

 //Constructor privado para que no se pueda hacer un new Configurador()
desde otro lugar que no sea dentro de la misma clase

 private Configurador() {}

 //Método estático encapsulador para acceder a la instancia única

public static Configurador getConfig() {
 if (config== null) {
 config= new Configurador();
 }
 return config;
 }
}
```

A pesar de que este código valida si existe una instancia antes de crearla, puede provocar un error si hay dos usuarios ejecutando el método al mismo tiempo, ya que ambos usuarios podrían entrar a `if(config==null)`, creando una instancia del Configurador para cada uno.

Cuando trabajamos con Java, cada vez que se ejecuta un método o algoritmo se crea un "hilo" y este se usará hasta que termine la ejecución. Hay algunos algoritmos que tardan milésimas de segundo y otros que duran meses ejecutándose, todo depende de su complejidad.

Por defecto, en aplicaciones online cada usuario genera su propio hilo al ejecutar un método, esto permite que la aplicación no se bloquee con un cuello de botella si dos usuarios quieren ejecutar el mismo método.

Synchronized

Cuando hablamos de Singleton, trabajaremos de una manera diferente a la que tienen por defecto los hilos. En el ejemplo anterior, necesitamos que la ejecución del método `getConfig()` se haga de manera sincronizada y así evitar tener hilos en paralelo al crear dos instancias de la misma clase. Para esto, utilizaremos la palabra reservada "synchronized".

```
public class Configurador{
 private static Configurador config;
 private Configurador() {}

public static Configurador getConfig() {
  if (config == null) {

 synchronized(Configurador.class) {

  if (config == null) {
   config = new Configurador();
   System.out.println("Instancia creada");
    }
  }
}
  System.out.println("Llamada al Configurador");
  return config
}
```

En el código anterior, podemos ver que si no existe una instancia `config == null` , se procede a iniciar una porción de código que está sincronizada mediante una sentencia **if**. Esto con la finalidad de que si hay dos o más usuarios tratando de acceder al mismo método, el primer usuario bloqueará el acceso al segundo. Para demostrar este ejemplo, podemos llamar al método `getConfig()` 2 veces en la clase `Main` y corremos nuestro programa.

```
public class Main{
public static void main(String[] args){
 Configurador.getConfig();
 Configurador.getConfig();
}
```

La consola nos arrojará como resultado que la instancia se crea con la primera persona y esta no se vuelve a crear cuando la segunda persona acceda a la validación de la instancia.

```
--------------------------
Impresión en pantalla:

Instancia Creada
Llamada al configurador
Llamada al configurador
```

Cierre

Hemos visto varios conceptos asociados a la Programación Orientada a Objetos, estos son la base que debe tener todo programador/a a la hora de realizar código. Sin embargo, existen otros principios que también pertenecen a las buenas prácticas del mundo del desarrollo.