

Universidad de Las Américas
Facultad de Ingenierías y Ciencias Agropecuarias
Ingeniería de Software
Informe de laboratorio

1. DATOS DEL ALUMNO: Camila Cabrera

2. TEMA DE LA PRÁCTICA: Implementación Patrón Singleton

3. OBJETIVO DE LA PRÁCTICA

- Realizar el ejercicio propuesto de implementación del Patrón Singleton para mejorar la comprensión de su aplicación.

4. OBJETIVOS ESPECÍFICOS

- Leer y comprender los principios y características del patrón Singleton.
- Inicializar Visual Studio Code para proyectos de java
- Crear un proyecto de java en visual Studio code
- Implementar la documentación para el proyecto ejemplo del patrón Singleton

5. MATERIALES Y MÉTODOS

Materiales:

- Visual studio code
- JDK 17
- Guía práctica de Patrón Singleton

Métodos:

1. Validación del JDK 17.
2. Configuración de Visual Studio Code para proyectos de Java.
3. Creación del proyecto Java sin herramientas de construcción del proyecto.
4. Implementación de la guía práctica del patrón Singleton.

5. Registro de hallazgos durante el proceso de implementación de la documentación.

6. DESARROLLO DE LA PRÁCTICA Y RESULTADOS

Marco Teórico

“Singleton es un patrón de diseño creacional que nos permite asegurarnos de que una clase tenga una única instancia, a la vez que proporciona un punto de acceso global a dicha instancia” (Refactoring guru, s.f.). Este patrón de diseño se aplica cuando se necesita garantizar que una clase tenga una única instancia y cuando se requiere proporcionar un punto de acceso global a dicha instancia. Este patrón garantiza que cuando se ha creado un objeto y se lo vuelva a llamar se obtiene el ya creado.

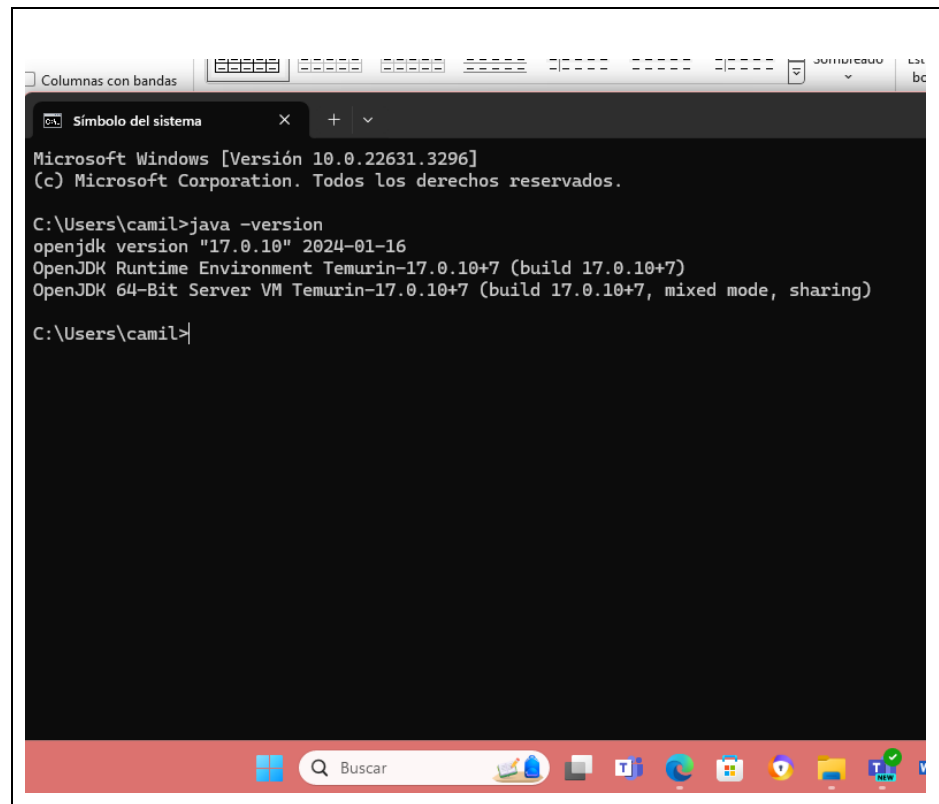
Para lograr este objetivo, es necesario hacer privado al constructor de la clase, y agregar una variable que guarde un objeto de esa clase. Luego se define un método estático que actúe como constructor. En este método se invoca al constructor privado para la creación del objeto y se guarda en el campo estático de la clase, si ya está creado el objeto se devuelve el objeto almacenado en caché.

Las ventajas de utilizar este patrón es que se puede tener la certeza que la clase tiene una única instancia, y tiene un punto de acceso global. Por ende, la inicialización del objeto solo ocurre la primera vez que se lo invoque. Sin embargo, hay que tener cuidado cuando se trabaja en un entorno con múltiples hilos de ejecución porque se crea el objeto varias veces, por ende, se debe modificar la estructura del patrón Singleton.

Desarrollo de la práctica

- **Validación del JDK 17:**
 - Para verificar la versión del JDK se comprueba con el comando “java -version” en la terminal del computador.

Imagen #1: Validación versión JDK



The image shows a Windows command prompt window titled 'Símbolo del sistema'. The command 'java -version' has been executed, resulting in the following output:

```
Microsoft Windows [Versión 10.0.22631.3296]
(c) Microsoft Corporation. Todos los derechos reservados.

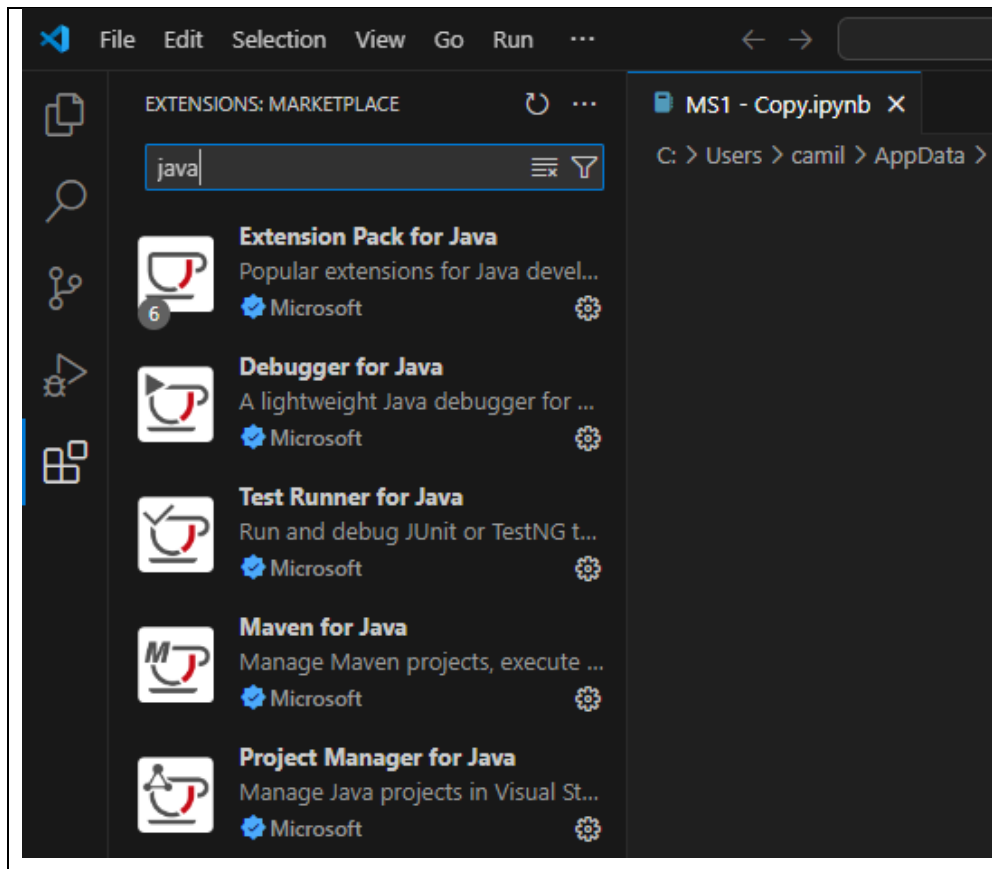
C:\Users\camil>java -version
openjdk version "17.0.10" 2024-01-16
OpenJDK Runtime Environment Temurin-17.0.10+7 (build 17.0.10+7)
OpenJDK 64-Bit Server VM Temurin-17.0.10+7 (build 17.0.10+7, mixed mode, sharing)

C:\Users\camil>
```

Fuente: Elaboración propia

- **2. Configuración de Visual Studio Code para proyectos de Java.**
 - En el IDE visual studio code se debe instalar o validar la dependencia para Java (Imagen 2) .

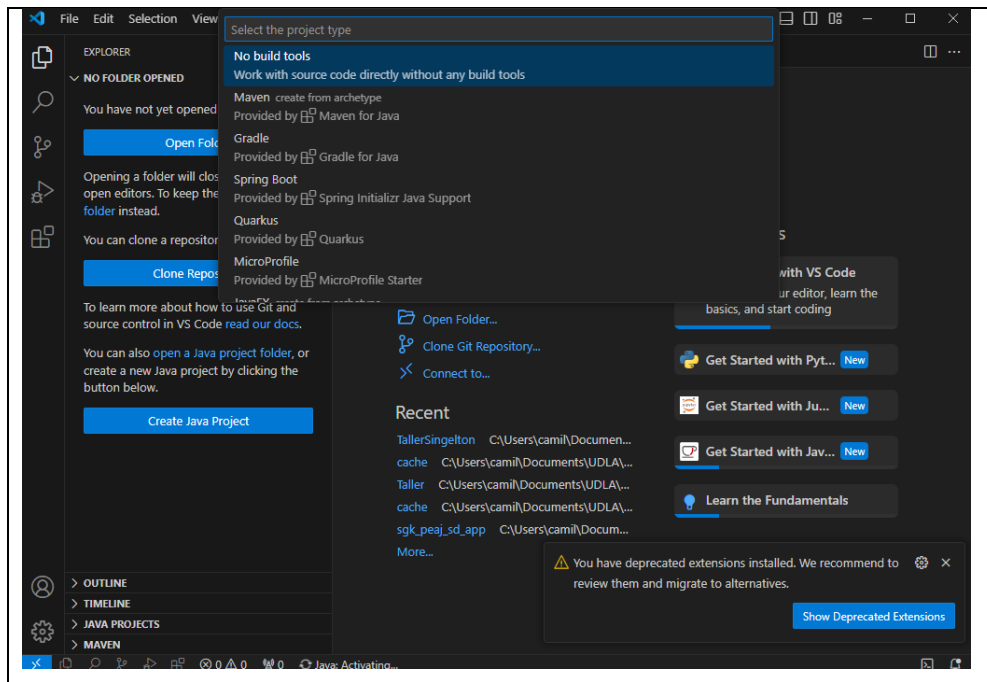
Imagen #2: Dependencia de Java



Fuente: Elaboración propia

- **Creación del proyecto Java sin herramientas de construcción del proyecto.**
 - Se inicializo el proyecto de Java escogiendo la opción sin herramientas de construcción.

Imagen #3: Dependencia de Java



- **Implementación de la guía práctica del patrón Singleton.**
 - Se siguió la guía práctica de Patron Singleton obtenida en el siguiente link: [Singleton en Java / Patrones de diseño \(refactoring.guru\)](https://refactoring.guru/design-patterns/singleton/java) para la elaboración del código.
 - La práctica de consistió en tres escenarios posibles Singleton ingenuo (hilo único), Singleton ingenuo (multihilo) y Singleton con seguridad en los hilos y carga diferida.

SINGLETON INGENUO (HILO ÚNICO)

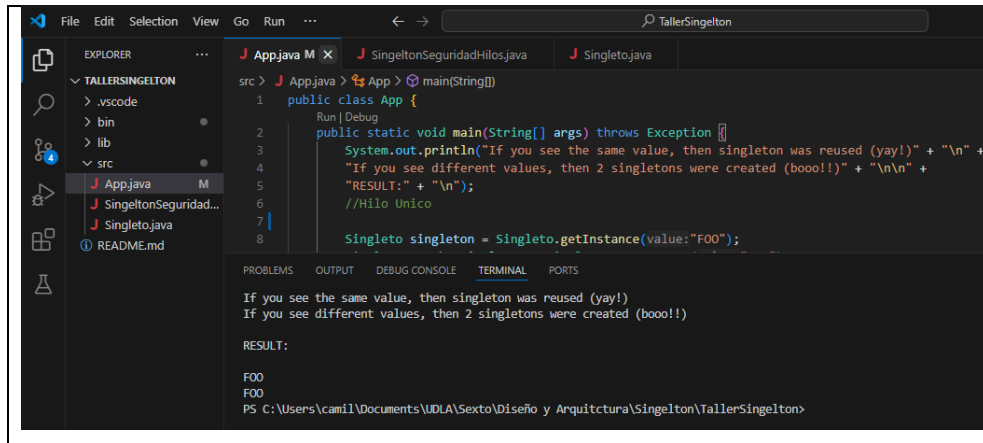
Clase: Singleton

- Se creó la clase llamada “Singleton” que cuenta con el atributo string value, y una variable privada estática de tipo Singleton, para almacenar la instancia creada.
- El constructor es de tipo privado y se simula la latencia en la creación del objeto.
- Se estableció un método “getInstance” para obtener la instancia única.

Clase: Principal

- En el método main se llama dos veces al método de la clase Singleton.getInstance y luego se imprime el valor. En este caso la primera instancia se llama FOO y la segunda BAR, por principios del patrón la instancia única será FOO, y esa es la que se imprime.

Imagen #4: Resultado ejecución



```
src > J App.java > App > main(String[])
1 public class App {
2     Run | Debug
3     public static void main(String[] args) throws Exception {
4         System.out.println("If you see the same value, then singleton was reused (yay!)") + "\n" +
5         "If you see different values, then 2 singletons were created (boooo!!)" + "\n\n" +
6         "RESULT:" + "\n");
7         //Hilo Unico
8         Singleton singleton = Singleton.getInstance(value:"FOO");

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
If you see the same value, then singleton was reused (yay!)
If you see different values, then 2 singletons were created (boooo!!)

RESULT:
FOO
BAR
PS C:\Users\camil\Documents\UDLA\Sexto\Diseño y Arquitectura\Singleton\TallerSingleton>
```

SINGLETON INGENUO (MULTIHILO)

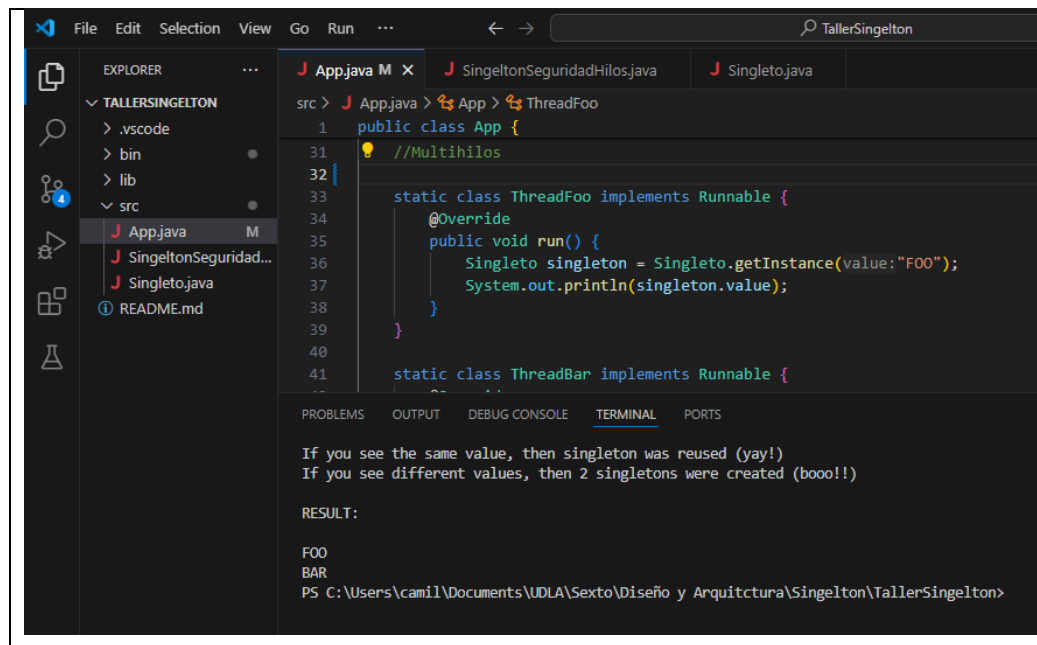
Clase: Singleton

- Se mantuvo la clase tal cual fue la creada para el anterior escenario.

Clase: Principal

- En el método main, programa crea dos subprocesos, cada uno de los cuales intenta acceder a la misma instancia de una clase Singleton.
- Se definen dos métodos que implementan la interfaz Runnable, que representa una tarea que se puede ejecutar en un subproceso.
- En el método run() de cada clase, se accede a la instancia de la clase Singleton mediante el método getInstance() y se imprime el valor del atributo value de esa instancia.
- El resultado de la ejecución imprime FOO y BAR ya que cada hilo toma una instancia de objeto singleton.

Imagen #4: Resultado ejecución



SINGLETON INGENUO (MULTIHILO)

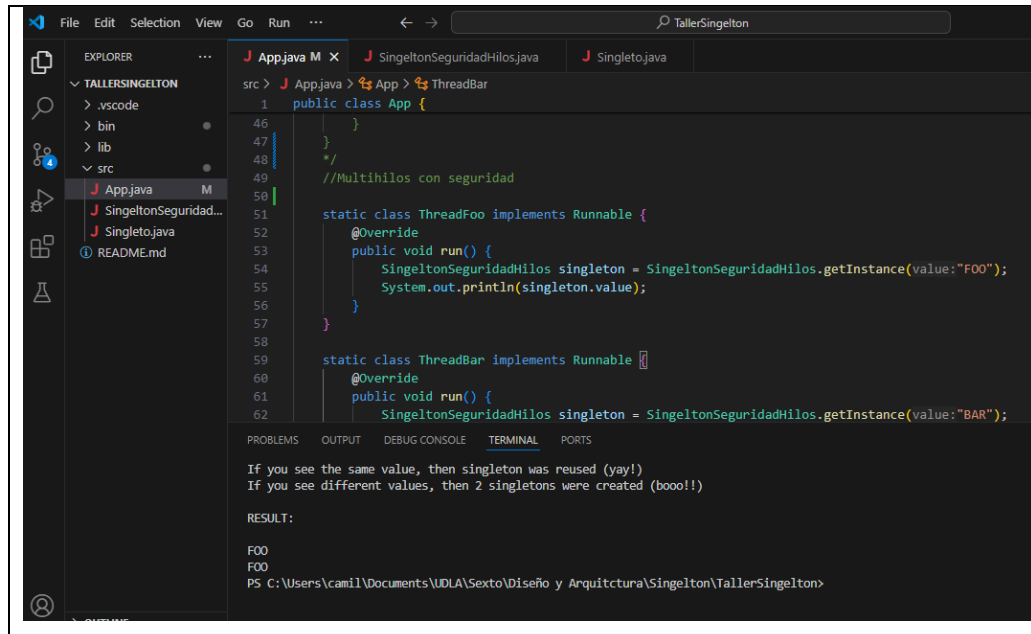
Clase: Singleton

- Se creo la clase llamada “SingletonSeguridadHilos” que cuenta como atributo un string value, y una variable privada estática volátil de tipo SingletonSeguridadHilos, para almacenar la instancia creada.
- El constructor es de tipo privado.
- Se estableció un método “getInstance” para obtener la instancia única. Este, utiliza el patrón de bloqueo de doble verificación (double-checked locking) para garantizar que solo se cree una instancia si aún no existe. En la primera validación, se asigna la instancia actual a la variable local result. Si ya existe una instancia, se devuelve esa instancia. Si no existe una instancia, se accede a un bloque sincronizado utilizando synchronized para evitar que múltiples hilos creen instancias simultáneamente. Dentro del bloque sincronizado, se realiza una verificación adicional para garantizar que solo se cree una instancia si aún no existe. Si es así, se crea una nueva instancia utilizando el valor proporcionado y se asigna a la variable instance.
- Finalmente, se devuelve la instancia creada o existente.

Clase: Principal

- Se utiliza la misma lógica de creación del método main del escenario anterior. Pero esta vez el resultado es diferente ya que solo se crea una instancia del objeto.

Imagen #5: Resultado ejecución



```
File Edit Selection View Go Run ...
TallerSingleton

EXPLORER
TALLER SINGLETON
  .vscode
  bin
  lib
  src
    App.java
    SingletonSeguridadHilos.java
    Singleton.java
    README.md

App.java
1 public class App {
46
47
48
49 //Multihilos con seguridad
50
51 static class ThreadFoo implements Runnable {
52   @Override
53   public void run() {
54     SingletonSeguridadHilos singleton = SingletonSeguridadHilos.getInstance(value:"FOO");
55     System.out.println(singleton.value);
56   }
57 }
58
59 static class ThreadBar implements Runnable {
60   @Override
61   public void run() {
62     SingletonSeguridadHilos singleton = SingletonSeguridadHilos.getInstance(value:"BAR");
63   }
64 }
65 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
If you see the same value, then singleton was reused (yay!)
If you see different values, then 2 singletons were created (boooo!!)

RESULT:
FOO
FOO
PS C:\Users\camil\Documents\UDLA\Sexto\Diseño y Arquitectura\Singleton\TallerSingleton>
```

- **Pruebas y Evaluación de los Resultados:**

- Como resultado se pudo cumplir con la guía de la práctica y comprender cómo se puede instanciar un solo objeto de una clase cuando se usa el patrón Singleton. Además, se abarcó el inconveniente de la creación de una única instancia cuando se utiliza varios hilos, y como se puede lograr la creación de una sola.

7. OPINIÓN PERSONAL

- Ya había tenido la oportunidad de trabajar en un entorno real con el patrón de diseño singleton, por tanto, la primera parte de la práctica no me resultó especialmente compleja. Sin embargo, la segunda parte al trabajar con subprocesos múltiples me pareció más complejo e importante de entender. Comprender este aspecto del Patrón singleton, exploro un área que ni

siquiera había considerado que podía existir, que fue la concurrencia en Java y las soluciones para abordarlos.

- En mi opinión usar este patrón en estos escenarios de prueba me han hecho reafirmar que, para poder aplicar un patrón de diseño, siempre se debe comprender cómo y cuándo aplicar el patrón para desarrollar sistemas robustos y eficientes. Por ejemplo, si no se tenía en consideración que el sistema va a utilizar subprocesos, entonces aplicar el primer enfoque daría un resultado erróneo ya que por cada proceso se instanciaría un nuevo objeto
- El patrón singleton en base a la lectura es uno de los patrones más comunes. Creo que se da porque es muy común el problema de instanciar solo una vez un objeto de una clase. Por ejemplo, en mi caso he visto que lo aplican a conexiones con bases de datos o web services. Inclusive en un proyecto en .net core se establece en la inyección de dependencia que el objeto será creado como singleton para no incurrir en la creación incensaría de conexiones.
- La parte mas enriquecedora de la práctica fue la importancia de abordar la concurrencia en la implementación del patrón y la implementación del bloqueo de doble verificación, además el poder explorar otros códigos en los diferentes lenguajes y ver que dependiendo del lenguaje y sus capacidades se da diferente enfoque para el mismo fin.

8. ANEXOS

Link Github del proyecto: [CamilaACT/LaboratorioSingleton \(github.com\)](https://github.com/CamilaACT/LaboratorioSingleton)

9. BIBLIOGRAFÍA

Refactoring guru. (s.f.). *Refactoring guru*. Obtenido de Singleton:
<https://refactoring.guru/es/design-patterns/singleton>