UT5 – UTILIZACIÓN AVANZADA DE CLASES

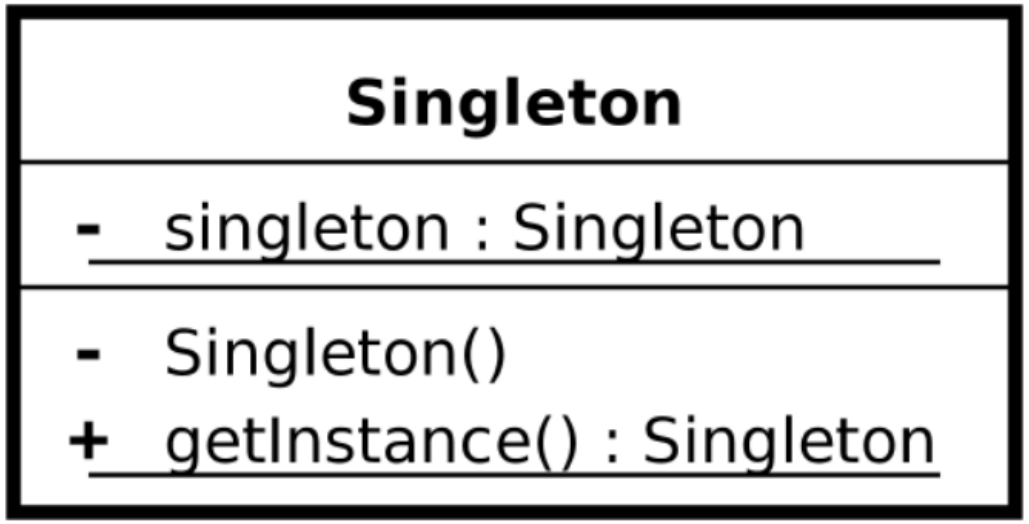
PATRÓN SINGLETON

Tipos de clases según su función

- Java puede diferenciar los tipos de clases según su función:
  - Clase Modelo: representan objetos o hechos que existen, suelen tener getter, setter, toString (coche, pez…)
  - Clase Servicio: implementan la lógica de negocio. Métodos públicos y atributos privados.
  - Clase Auxiliar: operaciones auxiliares, suelen tener los métodos estáticos.
  - Clase Main/Principal: la entrada de la aplicación. Suelen tener un método estático main.
  - Clase Test: orientados a hacer pruebas unitarias con Junit.

Clases SINGLETON

- ***Singleton*** es un patrón de diseño para clases Servicio y sirve para poder tener una clase de la cual **solamente queremos tener una instancia (manejadores, servicios, …).**
- Es útil en ciertas clases donde su única instancia es compartida en todo el código **como si de una variable global se tratase.**
- Si el método que permite obtener la referencia a la instancia es un método estático la instancia es como si fuese una variable global dado que es posible acceder a ella desde cualquier parte del programa siempre que los permitan **los modificadores de acceso.**



Clases SINGLETON – formas de implementar

- **Forma tradicional**. Para implementarla, podemos seguir los siguientes pasos:
  1. Definir una variable estática privada para guardar la referencia de la única instancia.
  2. Definir un único constructor, como privado, para evitar que el resto de clases puedan crear más instanciaciones.
  3. Obtener siempre la instancia a través de un método estático que valida si no ha sido creada con anterioridad y devuelve la referencia si ya existe la instancia.

Clases SINGLETON – formas de implementar

Forma tradicional

Este implementación , aunque es LAZY ya que la instancia no se crea hasta que se realiza la primera solicitud su inconveniente es que no THREAD-SAFE si varios hilos intentan obtener una instancia cuando aún no hay ninguna creada

```
public class Singleton {

    private static final Singleton instance;

    private Singleton() {}

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

Clases SINGLETON – formas de implementar

Método sincronizado

Implementa propiedad TREAD-SAFE, el método *synchronized*

Inconveniente hace que el método para obtener la instancia sea más lento debido a la propia sincronización y a la contención que se produce si hay múltiples hilos en ejecución que hacen uso del método

```
public class Singleton {

    private static final Singleton instance;

    private Singleton() {}

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

Clases SINGLETON – formas de implementar

Variable estática final

Implementa propiedad TREAD-SAFE

Inconveniente no es LAZY ya que la instancia se crea cuando se inicializa la clase antes de que se haga el primer uso

```
public class Singleton {

    private static final Singleton instance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return instance;
    }
}
```

Clases SINGLETON – formas de implementar

Clase inner

Implementa propiedad TREAD-SAFE y LAZY

Lo que hace es utilizar una clase anidada o inner para mantener la referencia a la instancia de la clase que la contiene

```
class Singleton {

   private static class SingletonHolder {
      public static Singleton instance = new Singleton();
   }

   private Singleton() {}

   public static Singleton getInstance() {
       return SingletonHolder.instance;
   }
}
```

El propio lenguaje java por la inicialización de las clases y propiedades static garantiza que es THREAD-SAFE

Clases SINGLETON – formas de implementar

Las Clases enum

Son por definición clases cuyos enumerados son singleton y thread-safe

```
public enum Singleton {

    INSTANCE;

    ...
}
```

La Particularidad de los ***enum*** es que no se pueden crear nuevas instancias adicionales a las que se definan en el código en tipo de compilación, el número de instancias es constante, por lo demás los enum al igual que las clases pueden implementar interfaces, declarar métodos y constructores de usuo en el propio enum

Conclusión

- Si el Singleton debe extender una clase la mejor opción de crearlo es con la clase inner aunque la opción del enumerado es válida.
- Java implementa un ejemplo de Patrón Singleton siguiendo esta opción.
- La forma tradicional sigue siendo muy utilizada con numerosos ejemplos de código en artículos y es posible utilizarla, sin embargo, dadas sus desventajas es preferible utilizar alguna de las dos aconsejadas.

```
public class NewSingleton {

    private NewSingleton() {
    }

    public static NewSingleton getInstance() {
        return NewSingletonHolder.INSTANCE;
    }

    private static class NewSingletonHolder {

        private static final NewSingleton INSTANCE = new NewSingleton();
    }
}
```

Ejemplo


```
public class MiServicioSingleton {

    // la clase inner(clase anidada) para mantener la referencia
    // a la instancia de la clase que la contiene
    // es donde se define la variable estática y se crea la instancia

    private static class MiServicioSingletonPoseedor {

        public static final MiServicioSingleton INSTANCE = new MiServicioSingleton();
    }
    // define el constructor privado
    private MiServicioSingleton() {

    }
    // obtener la instancia a través de un método estático
    public static MiServicioSingleton getInstance() {

        return MiServicioSingletonPoseedor.INSTANCE;
    }
```

Clases SINGLETON


```
public class Principal {
    public static void main(String[] args) {
        //Error
        //MiServicioSingleton s=new MiServicioSingleton();

        MiServicioSingleton s=MiServicioSingleton.getInstance();

        MiServicioSingleton s2=MiServicioSingleton.getInstance();

        System.out.println(s);
        System.out.println(s2);
    }
}
```

Ejecución:

Misma instancia

```
cic.ejemplos05.singleton.MiServicioSingleton@3fee733d
cic.ejemplos05.singleton.MiServicioSingleton@3fee733d
------------------------------------------------------------------------
```

Hoja05_Singleton_01