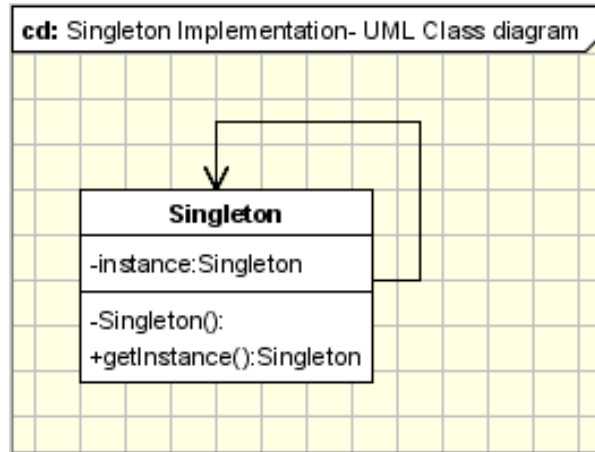


Singleton

Diagrama de clases e interfaces



Intención

- Asegurar de que sólo se puede crear una instancia de la clase.
- Proporcionar a otras clases un objeto con visibilidad global.

Motivación

A veces es importante que sólo exista una instancia de una clase. Las situaciones más habituales de aplicación de este patrón son aquellas en las que necesitamos:

- Una clase para controlar el acceso a un recurso físico único. Por ejemplo, si creamos una clase para gestionar ficheros nos bastará con tener un única instancia.
- Una clase que contenga ciertos datos que deben estar disponibles para el resto de objetos de la aplicación, es decir, necesitamos que un objeto tenga visibilidad global. Por ejemplo, en una aplicación de escritorio es habitual que por motivos de eficiencia de recursos sólo permitamos una conexión de base de datos por cada programa cliente.

El patrón Singleton es uno de los patrones de diseño más simples que hay: sólo interviene una clase -la clase Singleton-, la cual es responsable de instanciarse a sí misma, asegurándose de que no se cree más de un objeto de la clase. Esto implica que el constructor tiene que ser privado para evitar que las clases clientes puedan crear instancias. Al no haber un constructor accesible desde el exterior de la clase, para que las clases cliente puedan acceder a la instancia, la clase Singleton debe proporcionar un método público (un punto de acceso global) a dicha instancia. Por tanto, toda la aplicación utilizará siempre la misma instancia, la cual podrá invocarse desde cualquier parte.

Implementación

La implementación presenta las siguientes restricciones en la clase Singleton:

- Un miembro estático para almacenar la única instancia que se puede crear. Es estático para poder ser accedido por el método estático getInstance() que veremos a continuación (un método estático no puede acceder a las variables de instancia, sólo a las de clase).
- Un constructor privado para impedir la creación de instancias desde el exterior de la clase. Un efecto de declarar el constructor privado y no disponer de ningún otro público es que la clase Singleton no se podrá extender.
- Un método público y estático, normalmente llamado getInstance(), que retorne la referencia del miembro estático.

El patrón Singleton define una operación getInstance() que retorna la única instancia de la clase a las clases cliente. El método getInstance() es responsable de crear la instancia en caso de que aún no exista y devolverla.

```
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        ...
    }

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

```

        return instance;
    }

    ...

    public void hacerAlgo() {
        ...
    }
}

```

Podemos observar en el código anterior que el método `getInstance()` asegura que sólo se crea una instancia de la clase. El constructor no debe ser accesible desde el exterior de la clase para asegurar que la única forma de crear instancias de la clase es a través del método `getInstance()`.

Como se ha dicho anteriormente, el método `getInstance()` proporciona un punto de acceso global a la instancia. Si ahora para una clase cliente fuese de interés invocar al método `hacerAlgo()`, tendría que realizar la siguiente llamada:

```
Singleton.getInstance().hacerAlgo();
```

Aplicabilidad y Ejemplos

De acuerdo con su definición este patrón se utiliza cuando debe haber exactamente una instancia de una clase y se necesita que la instancia sea accesible globalmente. A continuación se enumeran algunas situaciones donde el patrón Singleton es adecuado:

Ejemplo 1 - Clases Logger

El patrón Singleton se utiliza en el diseño de las clases logger. Una clase logger se implementa normalmente como un Singleton, proporcionando servicios de log a cualquier clase de la aplicación sin tener que crear un objeto logger cada vez que se lleva a cabo una nueva operación de log.

Ejemplo 2 - Clases de configuración

Otro uso de este patrón es diseñar clases que contienen las opciones de configuración de una aplicación (idioma, ruta de ficheros, etc). Al implementar una clase como Singleton no sólo nos beneficiamos de disponer de un objeto con visibilidad global,

sino que además el objeto actúa como una caché de los datos de configuración, ya que el Singleton los almacenó en su estructura interna la primera vez que se llamó al método getInstance(). Este comportamiento de caché es muy interesante, dado que los datos de configuración normalmente están almacenados en ficheros o bases de datos, cuyo tiempo de acceso es muchísimo más lento que el acceso a un objeto de la memoria. Por tanto, nos evitamos la recarga de los datos desde la fuente original.

Ejemplo 3 - Accediendo a los recursos en modo compartido

En este ejemplo queremos utilizar el patrón en una aplicación que necesita trabajar con el puerto serie del ordenador. Supongamos que la aplicación se ejecuta en un entorno multi-hilo en el que las clases compiten por acceder al puerto serie. En este caso, se puede utilizar un objeto Singleton en el que las operaciones relativas al puerto serie utilicen sincronización.

Ejemplo 4 - Fábricas (factorías) implementadas como Singleton's

Supongamos que estamos diseñando una aplicación multi-hilo y hemos implementado un factoría (otro tipo de patrón) para generar nuevos objetos de diferente tipo (Cuenta, Cliente, Direccion), cada uno con su identificador de objeto (id). Si la factoría es utilizada por hilos diferentes es posible que acabemos teniendo dos objetos con el mismo id. Podemos solucionar este problema diseñando la factoría como una clase Singleton. Es una práctica muy común combinar el patrón Singleton con el patrón Abstract Factory o el con el Factory Method.

Problemas específicos e implementación

Caso 1: En un entorno multi-hilo necesitamos una implementación segura (thread-safe) del patrón Singleton

Una implementación robusta del patrón Singleton debe trabajar correctamente bajo cualquier condición, por tanto tenemos que asegurarnos de que funciona bien en un entorno multi-hilo. El código fuente presentado anteriormente es una implementación thread-safe del patrón Singleton

Caso 2: Instanciación perezosa utilizando la técnica de doble bloqueo

La implementación anterior del patrón Singleton es thread-safe, aunque no es la mejor implementación multi-hilo posible teniendo en cuenta que el mecanismo de sincronización es muy costoso computacionalmente cuando el rendimiento es un factor decisivo en una aplicación. Realmente, la sincronización del método `getInstance()` no es necesaria después de haber creado la instancia del Singleton, puesto que en las sucesivas invocaciones a `getInstance()` sólo tenemos que devolverlo (no hay peligro de crear un nuevo objeto que mandara al traste la finalidad del Singleton).

Existe una optimización que consiste en:

1. Comprobar dentro de un bloque no sincronizado si la instancia Singleton es nula.
2. Si la condición es cierta se vuelve a comprobar dentro de un bloque sincronizado la condición de nulidad.
3. Si la condición es cierta, se procede a la creación de la instancia.
4. Esto se conoce como mecanismo de doble bloqueo.

Igual que en la implementación anterior, la instancia Singleton se crea cuando el método `getInstance()` es invocado por primera vez. Esto se llama instanciación perezosa y es una técnica que asegura que la instancia Singleton se crea sólo cuando es necesario, por lo que es más eficiente que el código presentado anteriormente.

```
// Instanciacion perezosa utilizando la técnica del doble bloqueo.
public class Singleton {
    private static Singleton instance;

    private Singleton() {
        System.out
            .println("Singleton(): inicializando la instancia");
    }

    public static Singleton getInstance() {
        if (instance == null) {
            synchronized (Singleton.class) {
                if (instance == null) {
                    System.out
                        .println("getInstance(): metodo "+"
                            " getInstance ejecutado por " +
                            "primera vez!");
                    instance = new Singleton();
                }
            }
        }
    }
}
```

```

    }
}

return instance;
}

public void hacerAlgo() {
    System.out.println("hacer algo(): el Singleton hace algo!");
}
}

```

Caso 3: Instanciación impaciente utilizando la técnica del campo estático inicializado

En la siguiente implementación, el objeto Singleton se crea en cuanto se carga la clase y no cuando se ejecuta por primera vez el método getInstance(). Dado que el atributo instance es estático (como en todas las implementaciones vistas), se aprovecha para crear la instancia Singleton y asignarla al atributo, con lo que ya tenemos el atributo inicializado. Notad que también se declara el atributo instance como 'final'. Esto es una buena práctica, ya que impide que la referencia pueda apuntar a otro objeto una vez inicializada (queremos que el objeto referenciado sea siempre el mismo, es decir, hacemos la referencia constante).

De esta forma no necesitamos sincronizar ninguna parte del código, puesto que en una inicialización de clase no se pueden producir conflictos de hilos compitiendo por crear instancias. Sencillamente, la clase se carga una vez lo que garantiza la unicidad del objeto.

```

// Instanciación impaciente utilizando la técnica del campo estático inicializado
public class Singleton {
    private final static Singleton instance = new Singleton();

    private Singleton() {
        System.out
            .println("Singleton(): inicializando la instancia");
    }

    public static Singleton getInstance() {
        return instance;
    }

    public void hacerAlgo() {
        System.out
            .println("hacer algo(): el Singleton hace algo!");
    }
}

```

Una variante de esta implementación sería utilizar un inicializador estático para la creación de la instancia -invocando al constructor- y la asignación al atributo 'instance'. Esto es necesario cuando el proceso de construcción del objeto es susceptible de lanzar alguna excepción, ya que dentro del inicializador estático podemos capturar cualquier excepción y relanzarla.

```
// Instanciación impaciente utilizando la técnica del inicializador
estático para inicializar el campo estático
public class Singleton {
    private final static Singleton instance;

    /*
     * INICIALIZADOR ESTÁTICO
     * Al cargar la clase se ejecuta este código
     * sin necesidad de que exista un objeto de la clase
     */
    static {
        instance=new Singleton();
    }

    private Singleton() {
        System.out
            .println("Singleton(): inicializando la instancia");
    }

    public static Singleton getInstance() {
        return instance;
    }

    public void hacerAlgo() {
        System.out
            .println("hacer algo(): el Singleton hace algo!");
    }
}
```

Caso 4: Constructor protegido

Con el objetivo de permitir la creación de subclases de la clase Singeton, es posible declarar el un constructor protegido (protected) en lugar de privado (private). Esta técnica tiene 2 inconvenientes que hace que la herencia en el Singleton no sea práctica:

- Si el constructor se declara como protegido, significa que cualquier clase en el mismo paquete puede invocar al constructor e instanciar la clase. Una posible solución para evitarlo sería crear el Singleton en un paquete específico, donde no hubiera ninguna otra clase.

- En las clases cliente, para utilizar la clase derivada en lugar de la Singleton tienen que cambiarse todas las llamadas a Singleton.getInstance() por NewSingleton.getInstance() -ya que los métodos estáticos no son polimórficos-.

Caso 5: Varias instancias Singleton debido a cargadores de clases (classloaders) diferentes

Si una clase -Singleton o no- es cargada por dos cargadores de clases diferentes, se tendrán dos clases diferentes en memoria, a pesar que ambos cargadores la cargaron del mismo paquete y con el mismo nombre. A partir de aquí, en cualquier momento pasaremos a tener dos instancias del Singleton.

Caso 6: Serialización

Si la clase Singleton implementa la interfaz java.io.Serializable, tenemos el problema de que cuando el Singleton se serialice (por ejemplo se guarde como un fichero en disco) y posteriormente se deserialice (se lea el fichero de disco), nos encontremos con dos instancias del Singleton, la actual y la recuperada por el sistema de serialización. A pesar de que el Singleton tenga un constructor privado, el mecanismo de Serialización accede de un modo especial que le permite crear instancias de la clase.

La siguiente implementación pone de manifiesto esta situación. Notad que son dos clase, el Singleton y una clase cliente:

Clase Singleton

```
import java.io.Serializable;

public class Singleton implements Serializable {
    private static final long serialVersionUID = 1L;
    private static final Singleton instance = new Singleton();

    private Singleton() {
        System.out
            .println("Singleton(): inicializando la instancia");
    }

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

Clase cliente

```
package creacionales.singleton.main;
```



```

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

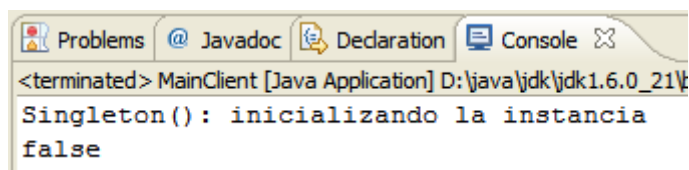
import creacionales.singleton.Singleton;

public class MainClient {

    public static void main(String[] args) throws Exception {
        Singleton INSTANCE = Singleton.getInstance();
        ObjectOutputStream oos =
            new ObjectOutputStream(
                new FileOutputStream("singleton.ser"));
        oos.writeObject(INSTANCE);
        oos.close();
        ObjectInputStream ois =
            new ObjectInputStream(
                new FileInputStream("singleton.ser"));
        Singleton test = (Singleton) ois.readObject();
        ois.close();
        System.out.println(test == INSTANCE);
    }
}

```

Ejemplo de ejecución:



Notad que el resultado es 'false', por tanto la clase cliente cuenta con dos instancias de Singleton!

Para evitar esta situación tendremos que implementar el método `readResolve()` de la interfaz `Serializable`. El método `readResolve()` nos permite que sea desechada la nueva instancia creada por el mecanismo de deserialización y que sea devuelta la instancia que realmente se almacenó en disco y que es la que tenemos referenciada por el atributo estático 'instance'. Para conseguir esto tan sólo tenemos que añadir el siguiente método en la clase `Singleton`:

```

...
//Este metodo es invocado inmediatamente despues de la
//deserializacion. Hacemos que retorne la instancia almacenada
//en el atributo estatico y la nueva instancia resultado de la
//deserializacion es desechada para el recolector de basura
private Object readResolve()
    throws java.io.ObjectStreamException {

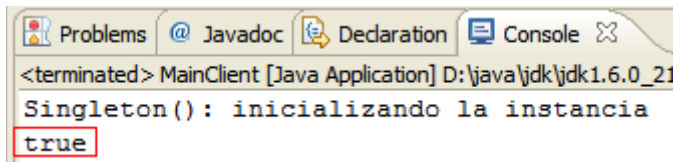
```

```

        return instance;
    }
    ...

```

Y ya está. No tenemos que hacer ninguna modificación en la clase cliente. Si ahora ejecutamos, obtenemos que las dos referencias apuntan a la misma instancia, por lo que el Singleton es consistente con su definición:



Caso 7: Los patrones Abstract Factory y Factory Method implementados como Singletons

En determinadas situaciones es conveniente que una factoría se haya implementado como un Singleton. Se comentó anteriormente en un ejemplo, el caso de dos hilos, donde cada hilo instanciaba un objeto de una factoría distinta y el resultado era que ambos objetos tenía el mismo ID.

Caso 8: Beneficios de utilizar una enum para implementar Singletons

A partir de Java 1.5 es posible utilizar una enum para implementar un Singleton. Las enum son por naturaleza Singletons y además solucionan de manera automática el problema de serialización visto anteriormente.

Veamos una posible implementación:

SingletonEnum.java (Notad que en lugar de una clase es una enum)

```

package creacionales.singleton;

import java.util.Arrays;

public enum SingletonEnum {
    instance;
    private final String[] bebidas = { "cerveza", "agua" };

    private SingletonEnum() {
        System.out
            .println("Singleton(): inicializando la instancia");
    }

    public void imprimirBebidas() {
        System.out.println(Arrays.toString(bebidas));
    }
}

```

```

    }

    public void hacerAlgo() {
        System.out
            .println("hacer algo(): el SingletonEnum hace algo!");
    }
}

```

MainClientEnum.java

```

package creacionales.singleton.main;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

import creacionales.singleton.SingletonEnum;

public class MainClientEnum {

    public static void main(String[] args) throws Exception {
        SingletonEnum INSTANCE = SingletonEnum.instance;

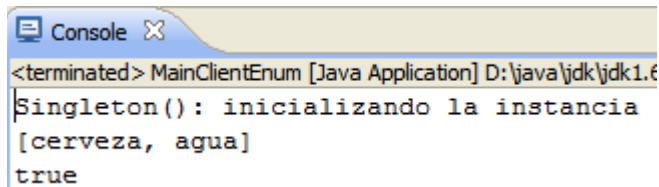
        INSTANCE.imprimirBebidas();

        ObjectOutputStream oos =
            new ObjectOutputStream(
                new FileOutputStream("singletonEnum.ser"));
        oos.writeObject(INSTANCE);
        oos.close();
        ObjectInputStream ois =
            new ObjectInputStream(
                new FileInputStream("singletonEnum.ser"));
        SingletonEnum test = (SingletonEnum) ois.readObject();
        ois.close();

        System.out.println(test == INSTANCE);
    }
}

```

Salida:



```

<terminated> MainClientEnum [Java Application] D:\java\jdk\jdk1.6
Singleton(): inicializando la instancia
[cerveza, agua]
true

```

Notad que el objeto apuntado por INSTANCE y por 'test' es el mismo después de deserializar, en cambio no hemos implementado readResolve() en la enum.

Desde luego es la opción más simple de utilizar, además de resolver los casos problemáticos que hemos visto.

Caso 9: Variante de Singleton con una clase interna

La variante vista anteriormente de instanciación perezosa con doble comprobación de bloqueo (caso 2) es demasiado verbosa. Con el cambio que supuso en Java 5 el nuevo modelo de memoria, es posible aplicar otra técnica que se podría considerar como el método estándar para implementar el patrón Singleton. Esta variante aprovecha la manera cómo se inicializan las clases en Java, y por lo tanto funcionará correctamente en todos los compiladores Java y máquinas virtuales.

Lo que caracteriza a esta variante es que utiliza una clase interna para inicializar el Singleton. La clase interna no es cargada por el ClassLoader hasta que es ejecutado el método `getInstance()`. Por lo tanto, esta solución es segura para entornos multihilo sin necesidad de utilizar estructuras especiales de lenguaje (uso de `volatile` o `synchronized`).

Veamos el código.

SingletonConClaseInterna.java

```
package creacionales.singleton;

import java.io.Serializable;

public class SingletonConClaseInterna implements Serializable {

    private static final long serialVersionUID = 1L;

    // Prevenir la instanciación desde el exterior de la clase
    private SingletonConClaseInterna() {
        System.out
            .println("SingletonInterna(): inicializando la instancia");
    }

    /*
     * La clase SingletonHolder se carga en la primera ejecución de
     * SingletonInterna.getInstance(), pero no antes
     */
    private static class SingletonHolder {
        private static final SingletonConClaseInterna instancia =
            new SingletonConClaseInterna();
    }

    /*
     * Metodo estatico publico que retorna la instancia. Notad que
     * lo que se
     * devuelve es el atributo estatico de la clase estatica
     */
}
```

```

        */
        public static SingletonConClaseInterna getInstance() {
            return SingletonHolder.instancia;
        }
    }
}

```

Desde luego es simple sin perder un ápice de potencia. Ahora veamos el código de una clase cliente que lo pruebe.

MainClientConClaseInterna.java

```

package creacionales.singleton.main;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;

import creacionales.singleton.SingletonConClaseInterna;

public class MainClientConClaseInterna {

    public static void main(String[] args) throws Exception {
        SingletonConClaseInterna INSTANCE =
            SingletonConClaseInterna.getInstance();

        ObjectOutputStream oos =
            new ObjectOutputStream(
                new
FileOutputStream("singletonConClaseInterna.ser"));
        oos.writeObject(INSTANCE);
        oos.close();
        ObjectInputStream ois =
            new ObjectInputStream(
                new
FileInputStream("singletonConClaseInterna.ser"));
        SingletonConClaseInterna test =
            (SingletonConClaseInterna) ois.readObject();
        ois.close();

        System.out.println(test == INSTANCE);
    }
}

```

Patrones relacionados

Muchos otros patrones se basan en algún grado en el patrón Singleton: Abstract Factory, Builder y Prototype son claros ejemplos.