**Singleton**

## **Intención**

**Singleton** es un patrón de diseño creativo que le permite asegurarse de que una clase tenga solo una instancia y proporcione un punto de acceso global a esta instancia.

## **Problema**

Singleton resuelve dos problemas en ese momento (violando el principio de responsabilidad única):

1. **Asegura que una clase tenga una sola instancia**. La razón más común para esto es controlar algunos recursos compartidos, por ejemplo, la base de datos.

Imagina que has creado un objeto y, después de un tiempo, intentas crear uno nuevo. En este caso, desearía recibir el objeto anterior, en lugar de crear una nueva instancia.

No se puede hacer con un constructor regular, ya que cada llamada de constructor **siempre** devuelve un nuevo objeto por diseño.

1. **Proporciona un punto de acceso global a esa instancia**. Suena como una variable global, ¿verdad? Pero no se puede hacer una variable global de solo lectura. Cualquiera que pueda acceder a él también puede reemplazar su valor.

Hay otra cara de este problema: no desea que el código que resuelve el problema anterior se disemine por todo su programa. Es mucho mejor tenerlo dentro de una clase, especialmente si el resto de su código ya depende de esa clase.

Tenga en cuenta que Singleton resuelve ambos problemas al mismo tiempo. Pero hoy en día el patrón se volvió tan popular que la gente llama a Singleton algo, incluso si resuelve solo uno de estos problemas.

## **Solución**

Todas las implementaciones de singleton comparten dos pasos comunes:

* Haciendo el constructor por defecto privado.
* Creando un método de creación estática que actuará como un constructor. Este método crea un objeto utilizando el constructor privado y lo guarda en una variable o campo estático. Todas las siguientes llamadas a este método devuelven el objeto en caché.

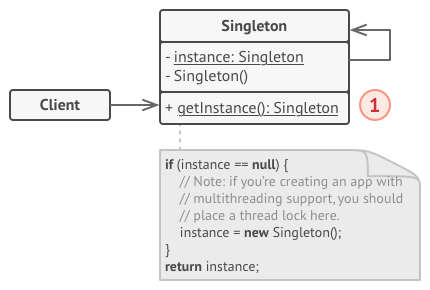
Singleton mantiene el código de producción de una sola instancia en un solo lugar, dentro del método de creación de una clase Singleton. Cualquier código de cliente que tenga acceso a la clase Singleton también tendrá acceso a su método de creación. Y eso nos da un punto de acceso global a la instancia de Singleton.

## **Analogía del mundo real**

**Gobierno**

El gobierno es un excelente ejemplo del patrón de Singleton. Un país puede tener un solo gobierno oficial. Independientemente de las identidades personales de las personas que forman gobiernos, el título "El gobierno de X" es un punto de acceso global que identifica al grupo de personas a cargo.

## **Estructura**



1. **Singleton** declara el método estático getInstance()que debe devolver la misma instancia de la clase Singleton .

El constructor de Singleton debe estar oculto del código del cliente. getInstance() Debería ser la única forma de crear y obtener objetos Singleton.

## **Pseudocódigo**

En este ejemplo, la clase de conexión de base de datos actúa como Singleton. Esta clase no tiene un constructor público, por lo que la única forma de obtener su objeto es llamar al getInstancemétodo. Este método almacena en caché el primer objeto creado y lo devuelve como resultado de todas las llamadas subsiguientes.

El patrón de diseño único garantiza que se creará la única instancia de su clase. Y proporciona un punto de acceso global a esta instancia: el getInstancemétodo estático .

**class** **Database** **is**

**private** **field** instance: Database

**static** **method** getInstance() **is**

**if** (**this**.instance == **null**) **then**

acquireThreadLock() **and** **then**

// Ensure that instance has not yet been

// initialized by other thread while this one

// has been waiting for the lock release.

**if** (**this**.instance == **null**) **then**

**this**.instance = **new** Database()

**return** **this**.instance

**private** **constructor** Database() **is**

// Some initialization code, such as the actual

// connection to a database server.

*// ...*

**public** **method** query(sql) **is**

// All database queries of an app will go through this

// methods. Therefore, you can place a throttling or

// caching logic here.

*// ...*

**class** **Application** **is**

**method** main() **is**

Database foo = Database.getInstance()

foo.query("SELECT ...")

*// ...*

Database bar = Database.getInstance()

bar.query("SELECT ...")

// The variable "bar" will contain the same object as

// the variable "foo".

## **Applicabilidad**

**Cuando el programa debe tener una sola instancia de una clase, disponible para todos los clientes. Por ejemplo, un único objeto de base de datos, compartido por diferentes partes del programa.**

Singleton oculta a los clientes todos los medios para crear nuevos objetos de su clase, excepto el método de creación especial. Este método crea un nuevo objeto o devuelve un objeto existente si se ha creado anteriormente.

**Cuando necesite un control más estricto sobre las variables globales.**

A diferencia de las variables globales, Singleton garantiza que solo hay una instancia de clase. Nadie, excepto el propio Singleton, puede reemplazar la instancia en caché.

Además, Singleton te permite cambiar fácilmente esta restricción. Por ejemplo, para permitir cualquier número de instancias, solo tendrá que modificar el código en un solo lugar: dentro del getInstance()cuerpo.

## **Cómo implementar**

1. Agregue a la clase un campo estático privado que contendrá la instancia de singleton.
2. Declare el método de creación estática pública que se utilizará para recuperar la instancia de singleton.
3. Implementar "inicialización perezosa" dentro del método de creación. Debe crear una nueva instancia en la primera llamada y colocarla en el campo estático. El método debería devolver esa instancia en todas las llamadas subsiguientes.
4. Hacer constructor de clase privado.
5. En el código del cliente, reemplace todas las llamadas directas del constructor con llamadas al método de creación estática.

## **Pros y contras**

**Pros**

* Asegura que la clase tiene una sola instancia.
* Proporciona un punto de acceso global a esa instancia.
* Permite la inicialización diferida.

**Contras**

* Viola el *principio de responsabilidad única*.
* Máscaras de mal diseño.
* Requiere un tratamiento especial en un entorno multihilo.
* Requiere burlas infinitas en pruebas unitarias.

## **Ejemplo en C#**

#### **Program.cs:** Structural Example

﻿**using** System;

**using** System.Collections.Generic;

**using** System.Linq;

**using** System.Text;

**using** System.Threading.Tasks;

**namespace** **Singleton**

{

**class** **Program**

{

**static** **void** Main(**string**[] args)

{

Client client = **new** Client();

client.ClientCode();

}

}

**class** **Client**

{

**public** **void** ClientCode()

{

Singleton s1 = Singleton.getInstance();

Singleton s2 = Singleton.getInstance();

**if**(s1 == s2)

{

Console.WriteLine("Singleton works, both variables contain the same instance.");

}

**else**

{

Console.WriteLine("Singleton failed, variables contain different instances.");

}

}

}

**class** **Singleton**

{

**private** **static** Singleton instance;

**private** **static** **object** obj = **new** **object**();

**private** Singleton()

{ }

**public** **static** Singleton getInstance()

{

**lock**(obj)

{

**if** (instance == **null**)

instance = **new** Singleton();

}

**return** instance;

}

}

}

#### **Output.txt:** Output

Singleton works, both variables contain the same instance.