# Un solo pato

Original: *Head First Design Patterns*, editorial **O'Reilly**

Resulta que tu empresa sigue con el tema de los **Patos**. Ahora el jefe quiere que añadas al programa una **Granja** de patos. Esta granja puede hacer diferentes cosas como, por ejemplo, consultar si hay patos dentro, cuántos huevos han puesto, añadir un pato nuevo, etc.

Ahora bien, el jefe nos dice que esa Granja **tiene que ser única** para todo el programa. No puede haber dos. Cada vez que el programa quiera hacer algo con esa Granja, independientemente desde dónde sea, tiene que ser siempre la misma Granja.

¿Ideas?

### Idea 1

Cada vez que hacemos un **new Granja ()** lo que sucede es que en tiempo de ejecución se está reservando en memoria un espacio para una nueva Granja; lo asignemos o no a una variable.

```
public voi crearGranjas (){

    // +1 Granja
    Granja granja = new Granja ();

    // +1 Granja
    Granja granja2 = new Granja ();

    // +1 Granja
    new Granja ();

}
```

Tal y como funciona la gestión de memoria de Java, esas tres granjas sólo son accesibles desde dentro de su ámbito, que en este caso, son las { } del método crearGranja (). Pero esto NO significa que se borren de memoria si sales de su ámbito. Simplemente no puedes acceder a ellas.

La memoria se libera automáticamente gracias al Recolector de Basura de Java. Cuando 'algo' en memoria ya no tiene variables que lo referencien, el recolector lo elimina. Lamentablemente, no sabemos cuándo se va a pasar el Recolector de Basura. Eso lo hace la Java Virtual Machine cuando lo necesita.

Por tanto, parece obvio que tenemos que hacer un único new Granja (), y asegurarnos de que esa Granja se pase a todo el programa todo el tiempo. Parece sencillo, ¿no?

```
public class Launcher () {

    private Granja granja;

    public Launcher (){
        this.granja = new Granja ();
    }

    public void init (){

        Gestion gestion = new Gestion (granja);

        Almacen almacen = new Almacen (granja);

        // -- etc...

    }

}
```

En este ejemplo, creamos Granja en el constructor de Launcher, y luego, cuando se llame a init (), instanciamos todos los demás objetos que necesitamos pasándole la referencia de la Granja.

¿Funcionaría?

Desde luego. Pero esta solución tiene muchas pegas. Para empezar, la más obvia, es que estás creando una Granja al inicio del programa y después se la estás pasando a todos los demás objetos. ¿Qué sucedería si por ejemplo Almacén no necesitase Granja más que el 3% de las veces? Tendrías una variable con algo creado que no usas. Una receta para el desastre. ¿No sería mejor crearla solamente cuando sea necesario?

Tampoco estás teniendo en cuenta que alguien podría hacer un new Granja () en otro punto del programa, ya sea por despiste o simplemente porque puede. Total, no hay nada que te impida instanciar una Granja si sólo vas a usarla un poquito… ¿verdad? Porque controlas lo que estás haciendo… ¿no?

A demás, tenemos un problema con la legibilidad y el mantenimiento del programa. Si en vez de una tuviésemos siete objetos diferentes con las mismas condiciones (ser únicos en el programa) tendríamos un código caótico con un montón de variables que estás pasando todo el rato y que a lo mejor ni utilizas. Constructores con 7 parámetros para nada. Esto es inmanejable a largo plazo.

Por tanto, esta idea podría servirnos para cosas pequeñas y muy concretas. Pero nada más. ¿Alguna solución mejor?

**Idea 2**

Podríamos hacer que Granja fuese **estática** (static). Algo declarado como estático en Java se coloca en una posición de memoria específica y se puede hacer accesible para todo el mundo, ya sea un atributo o un método de clase. Por tanto, cada vez que alguien quiera acceder a Granja, que acceda al atributo estático y ya.

Algo como esto:

```
public class Launcher () {

    public static Granja granja = new Granja();

    public void init (){

        Gestion gestion = new Gestion ();

        Almacen almacen = new Almacen ();

        // -- etc...

    }

}

// En cualquier momento podemos hacer esto...
Granja granjaUnica = Launcher.granja;
```

El problema que tenemos aquí es que no estamos respetando mucho el tema de la **encapsulación de código** (ver pdf anterior) al darle acceso a otros objetos a los atributos de Launcher. En este caso en concreto, podemos cambiar Granja con este simple código:

```
Launcher.granja = new Granja();
```

Lo que podría generar muchos errores en nuestro programa en un descuido. Encontrar la metedura de pata debuggeando puede llevar mucho tiempo. Podríamos intentar arreglar la clase añadiendo **final** a la variable estática, o haciéndola **private** y haciendo getters & setters estáticos… o yo qué sé. Fijo que se te ocurre algo.

… pero vamos a ver, seguro que este problema ya lo ha tenido alguien, ¿no? Pues eso. Al grano.

**Idea 3**

Vamos a ver qué patrón cumple nuestros requisitos. Para el caso de que queramos tener un único objeto de una clase en memoria y accesible para todo el programa, respetando la encapsulación, haciéndolo legible, mantenible, etc. la solución es el **Singleton Pattern**.

El más fácil de programar y entender. Aparentemente.

```
public class Granja () {

    private static Granja instance;

    // Privado, para que no me hagan new Granja()
    private Granja (){
        // Por si hay que inicializar algo
    }

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

    // Resto de métodos
}
```

La clase tiene una variable instance que es estática. La única forma de que se instancie es a través del método **getInstance ()**, que también es estático. Este método pregunta si instance es nula, y si lo es, hace el new y lo devuelve. En cualquier otro caso, simplemente devuelve instance. De esta forma podemos estar seguros de que solamente va a haber una Granja en todo el programa.

Cada vez que hacemos desde cualquier parte del programa:

```
Granja granja = Granja.getInstance();
```

La primera vez sí se hace new Granja (), pero las demás veces simplemente se devuelve esa Granja. Y no necesitas más que esta línea de código para obtener esa Granja única.

Esta solución es elegante, mantiene el código encapsulado, es limpia, fácil de mantener, etc.

La solución ferpecta…

**Problemas raros con el Singleton**

Excepto cuando tenemos un programa con **Hilos** (Threads). Para entender por completo el problema, tienes que entender primero lo que es un Hilo en Java, lo que es la Concurrencia y los problemas de Sincronización que acarrea trabajar con Hilos. Esto se ve en 2ndo en la asignatura de PSP.

Aquí la cosa se complica.

Supongamos que el jefe ha dicho que nuestro programa de patos va a estar funcionando para 20 empresas diferentes – los patos son un negocio – pero que Granja tiene que seguir siendo única. Aplicamos convencidos el Patrón a la clase Granja, y generamos un hilo para cada empresa. Arrancamos el programa y todo va genial… hasta que de repente falla.

¿Dónde está el error?

Los **Hilos** en Java se ejecutan en un orden que no podemos predecir. Esto lo decide la Java Virtual Machine por medio de una serie de factores que no vamos a desarrollar aquí. Lo que nos interesa es que si tú tienes 20 Hilos tratando de acceder a un mismo recurso (Granja) es posible que haya 2 tratando de entrar al mismo sitio a la vez. Este caso, la zona conflictiva es **getInstance ()**.

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

Imagínate que el Hilo 1 entra en getInstance (), pero acto seguido, el Hilo 2 entra en el mismo método. Podría suceder que ambos ejecutasen al mismo tiempo instance = new Granja (), generándose el objeto dos veces.

La forma de solucionarlo parece simple. Añadiendo la clave **synchronized** al getInstance (), forzamos a que solamente un hilo pueda acceder al método en cada instante de tiempo. Así nos libramos del problema.

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

Y ya está… ¿no?

**Más problemas raros con el Singleton**

Pues no. La sincronización de **Hilos** en Java es algo muy costoso en tiempo de ejecución. En el ejemplo anterior, en realidad, sólo hay un problema de sincronización la primera vez que se accede a getInstance (). Por tanto, el resto de las veces, no haría falta sincronizar.

¿Qué hacemos entonces?

Bien, pues aquí es cuando tenemos que pensar un poco en las diferentes posibilidades que se nos ofrecen.

1) Si no es algo crítico, o el rendimiento no es importante, **ignora** el asunto y deja el synchronized. Synchronized es la forma correcta y efectiva de solucionar estos problemas. Sólo recuerda que estás empeorando el rendimiento del programa en un factor 100. Algo que tardaba un segundo, ahora tarda 100 segundos. Y a veces esto no puede ser.

2) En vez de crear instance de forma **lazy** (hacer el new () cuando la necesitas al llamar getInstance ()) lo vamos a crear de forma **eager** (cuando la clase Granja se crea).

   Esto deja la clase tal que así:

```
public class Granja () {

    private static Granja instance = new Granja();

    // Privado, para que no me hagan new Granja()
    private Granja (){
        // Por si hay que inicializar algo
    }

    public static Granja getInstance (){
        return instance;
    }

    // Resto de métodos
}
```

   Esta idea funciona de maravilla cuando estás 100% seguro de que siempre vas a necesitar la Granja, y el rendimiento con el tema de los Hilos es importante. Esto no causa problemas porque la Java Virtual Machine genera una instancia de Granja (instance) en cuanto se carga la clase, por tanto, cuando un Hilo entra en getInstance (), ya tiene el objeto accesible y no importa cuántos más accedan a la vez.

3) Usemos **“doble-check-locking”**. De esta forma, primero miramos si la instancia se ha creado, y si no, entonces sincronizamos. Así, solucionamos el problema de antes.

```
public class Granja () {

    private volatile static Granja instance;

    // Privado, para que no me hagan new Granja()
    private Granja (){
        // Por si hay que inicializar algo
    }

    public static Granja getInstance (){
        if (instance == null){
            synchronized(Granja.class) {
                if (instance == null){
                    instance = new Granja();
                }
            }
        }
        return instance;
    }

    // Resto de métodos
}
```

Al añadir la clave **volatile**, nos aseguramos que los hilos que intenten acceder a instance lo hagan correctamente al ser inicializada en Granja. Fíjate también en getInstance (). La primera vez que se accede, se comprueba si instance existe. Después, se sincronizan los hilos (se hará sólo una vez) y después, comprobamos de nuevo que no sea null. Solamente entonces se instanciará instance.

Esto no funciona en Java 1.4 o anterior, pero soluciona los problemas de rendimiento con los hilos.

4) Usemos un **enum**. Solucionaría todos los problemas mencionados.

```
public enum Granja {

    INSTANCE;

    // Resto de métodos

}


// En cualquier momento podemos hacer esto...
Granja granja = Granja.INSTANCE
```

## Conclusión

Hay varias formas de solucionar el problema de tener una única instancia de una sola clase. En la mayoría de los casos, si no hay Hilos, no tienes que complicarte la vida y la primera solución es adecuada. Si tienes Hilos, dispones de varias soluciones posibles. Y la última, la que usa enum, es perfecta porque le da igual que haya Hilos y es muy fácil de leer. Aunque las versiones viejas de Java no tenían enum…

Lo importante de todas formas es que hayas entendido el problema, las bases de Singleton, y los problemas que están relacionados con cada una de sus implementaciones.

¡Buen trabajo!

## Ejercicio Propuesto

Crea una Granja usando Singleton. Añade algunos métodos a la Granja como, por ejemplo:

- AñadirPato () – Añade un pato a una lista en Granja
- QuitarPato () – Quita un pato de la lista
- ContarPatos () – Te dice cuántos patos hay

Crea ahora un programa que utilice Granja para añadir algunos patos con un menú de opciones para acostumbrarte a cómo se codifica con este patrón.

¿Ya? Bien. Ahora genera un programa que lance 20 Hilos. Cada Hilo va a despertar aleatoriamente cada 1-5 segundos y va a utilizar Granja. Cada vez que un Hilo haga algo, lo muestras en consola. Debería verse algo así como:

"Hilo 14 – Añadir Pato – Hay 7 patos en la Granja."

Prueba diferentes implementaciones para ver si efectivamente, al usar el **synchronized** se nota la caída de rendimiento del programa.

Si no lo notas, prueba a añadir muchos más Hilos.