

Patrón de diseño “Singleton”



Antes del problema concreto...

¿Alguna vez han necesitado una clase con exactamente una instancia, es decir un solo objeto?

Este tipo de clases se ocupan en todos lados:

- Bases de datos
- Interfaces gráficos
- Páginas de internet con sesión

¿Java nos permite eso?



¿Java nos permite eso?

Para instanciar un objeto en java se usa

new MiObjeto();

que no nos da ninguna restricción para crear nuevas instancias.



Y, ¿qué pasaría si otro objeto quisiera crear un MyObject? ¿Podría llamar a new en MyObject nuevamente?

Sí, por supuesto.




Entonces, mientras tengamos una clase, ¿siempre podemos crear una instancia una o más veces?

- Sí, solo si es una clase pública.

¿Y si no?

- Bueno, si no es una clase pública, solo las clases en el mismo paquete puede instanciarlo. Pero aún pueden crear una instancia más de una vez.

**¿Entonces como
logramos tener
únicamente una
instancia de un
objeto?**




Utilicemos los permisos de los métodos

¿Que pasa si ponemos los permisos de acceso al constructo por privados?

```
public MyClass {  
    private MyClass(){}  
}
```

¿Qué significa eso?

Supongo que es una clase que no se puede instanciar porque tiene un constructor privado.



¿hay algún objeto que pudiera utilizar el constructor privado?

Hmm, creo que el código en MyClass es el único código que podría llamarlo. Pero eso no tiene mucho sentido.



¿Que me pueden decir de la siguiente declaración?

```
public MyClass {  
    public static MyClass getInstance() {  
        return new MyClass();  
    }  
}
```

- MyClass es una clase con un método estático.
Podemos llamar al método estático así:

```
MyClass.getInstance();
```

¿Por qué usaste MyClass, en lugar de un nombre de objeto?

- getInstance () es un método estático; en otras palabras, es un método de clase. Por lo que debe usar el nombre de la clase para hacer referencia a un método estático.

¿Qué pasa si juntamos ambas ideas? ¿Ahora puedo crear una instancia de MyClass?

```
public MyClass {  
    private MyClass(){}  
  
    public static MyClass getInstance() {  
        return new MyClass();  
    }  
}
```


- Por lo que veo si se puede.

Entonces, ¿puedes pensar en una segunda forma de instanciar un objeto?

- `MyClass.getInstance();`

¿Puedes terminar el código para que solo se haya creado UNA instancia de MyClass?

Sí, eso creo ...



Let's rename MyClass
to Singleton.

We have a static
variable to hold our
one instance of the
class Singleton.


```
public class Singleton {  
    private static Singleton uniqueInstance;  
  
    // other useful instance variables here  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new Singleton();  
        }  
        return uniqueInstance;  
    }  
  
    // other useful methods here  
}
```

Our constructor is
declared private;
only Singleton can
instantiate this class!

The getInstance()
method gives us a way
to instantiate the class
and also to return an
instance of it

Of course, Singleton is
a normal class; it has
other useful instance
variables and methods.

¿Esto en que nos ayuda?

- Se puede crear un objeto llamando a `Singleton.getInstance()`;
 - El objeto se puede guardar con `Singleton si = Singleton.getInstance()`;
 - Todas las veces que se guarde o se use ese objeto será la misma instancia del objeto.
 - Solo se crea una instancia del objeto si se va a usar alguna vez.
- 

Regresemos a nuestra programación habitual

Problemática del día


- Willy Wonka nos a llamado, al parecer los Oompa Loompas no saben usar computadoras
- Todo el mundo sabe que todas las fábricas de chocolate modernas tienen calderas de chocolate controladas por computadora.
- El trabajo de la caldera es tomar chocolate y leche, hervirlos y luego pasarlos a la siguiente fase de hacer barras de chocolate.



Problemática del día

Esta es la clase de controlador para Wonka, Inc.'s fuerza industrial de Caldera de Chocolate.

A continuación te mostrare su código; notarás que han tratado de ser muy cuidadosos para asegurar que las cosas malas no sucedan, como drenar 500 galones de mezcla sin hervir, o llenar la caldera cuando ya está llena, o hervir una caldera vacía.




Diseño actual

```
public class ChocolateBoiler {  
    private boolean empty;  
    private boolean boiled;
```


```
    public ChocolateBoiler() {  
        empty = true;  
        boiled = false;  
    }
```

*This code is only started
when the boiler is empty!*



```
    public void fill() {  
        if (isEmpty()) {  
            empty = false;  
            boiled = false;  
            // fill the boiler with a milk/chocolate mixture  
        }  
    }
```

*To fill the boiler it must be
empty, and, once it's full, we set
the empty and boiled flags.*



```
public void drain() {  
    if (!isEmpty() && isBoiled()) {  
        // drain the boiled milk and chocolate  
        empty = true;  
    }  
}
```

↖ To drain the boiler, it must be full (non empty) and also boiled. Once it is drained we set empty back to true.

```
public void boil() {  
    if (!isEmpty() && !isBoiled()) {  
        // bring the contents to a boil  
        boiled = true;  
    }  
}
```

↖ To boil the mixture, the boiler has to be full and not already boiled. Once it's boiled we set the boiled flag to true.

```
public boolean isEmpty() {  
    return empty;  
}
```

```
public boolean isBoiled() {  
    return boiled;  
}
```

Problema concreto: Chocolate

Esta fábrica de chocolates nos pide arreglar el diseño de su sistema, pues tiene una clase para controlar la caldera del chocolate, pero en su sistema actualmente todas las clases pueden acceder y reinstanciar esa clase, por lo que se puede perder la referencia en el código.

Apliquemos lo aprendido

Dado lo visto antes del problema es fácil arreglar el diseño a algo que nos permita solucionar el problema.



```
public class ChocolateBoiler {
    private boolean empty;
    private boolean boiled;
    private static ChocolateBoiler uniqueInstance;

    private ChocolateBoiler() {
        empty = true;
        boiled = false;
    }

    public static ChocolateBoiler getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new ChocolateBoiler();
        }
        return uniqueInstance;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
            // fill the boiler with a milk/chocolate mixture
        }
    }

    // rest of ChocolateBoiler code...
}
```

Definición del patrón


El patrón de diseño “Singleton” se asegura de que una clase tiene solamente una instancia y provee de un punto de acceso global a esta.

El patrón Singleton garantiza que una clase tenga solo una instancia y proporciona un punto de acceso global a ella.



No hay grandes sorpresas allí. Pero, analizémoslo un poco más:

¿Qué está pasando realmente aquí? Estamos tomando una clase y dejándole administrar una sola instancia de sí mismo. También estamos evitando que cualquier otra clase cree una nueva instancia por sí misma. Para obtener una instancia, debe pasar por la clase en sí.



También proporcionamos un punto de acceso global a la instancia: siempre que necesite una instancia, simplemente consulte la clase y le devolverá la instancia única. Como ha visto, podemos implementar esto para que Singleton se cree de manera perezosa, lo que es especialmente importante para los objetos de uso intensivo de recursos.

Diagrama de clases (o clase)


The getInstance() method is static, which means it's a class method, so you can conveniently access this method from anywhere in your code using Singleton.getInstance(). That's just as easy as accessing a global variable, but we get benefits like lazy instantiation from the Singleton.

Singleton
static uniqueInstance
// Other useful Singleton data...
static getInstance()
// Other useful Singleton methods...

The uniqueInstance class variable holds our one and only instance of Singleton.

A class implementing the Singleton Pattern is more than a Singleton; it is a general purpose class with its own set of data and methods.


Problemas y soluciones con este patrón

- X** No puedes extender la clase del patrón Singleton, pues sus descendientes no pueden acceder al constructor de su padre.
 - ✓ Hay que decidir donde usar este patrón y si de verdad es mas conveniente restringir el número de instancias a extender la clase en un futuro.
- 



Otra vez con problemas XD

Parece que la caldera de chocolate nos ha decepcionado; a pesar de que mejoramos el código usando el Clásico Singleton, de alguna manera el método de llenado de `ChocolateBoiler()` pudo comenzar a llenar la caldera a pesar de que un lote de leche y chocolate ya estaba hirviendo. ¡Eso es 500 galones de leche derramada (y chocolate)! ¿¡Que pasó!? :'(

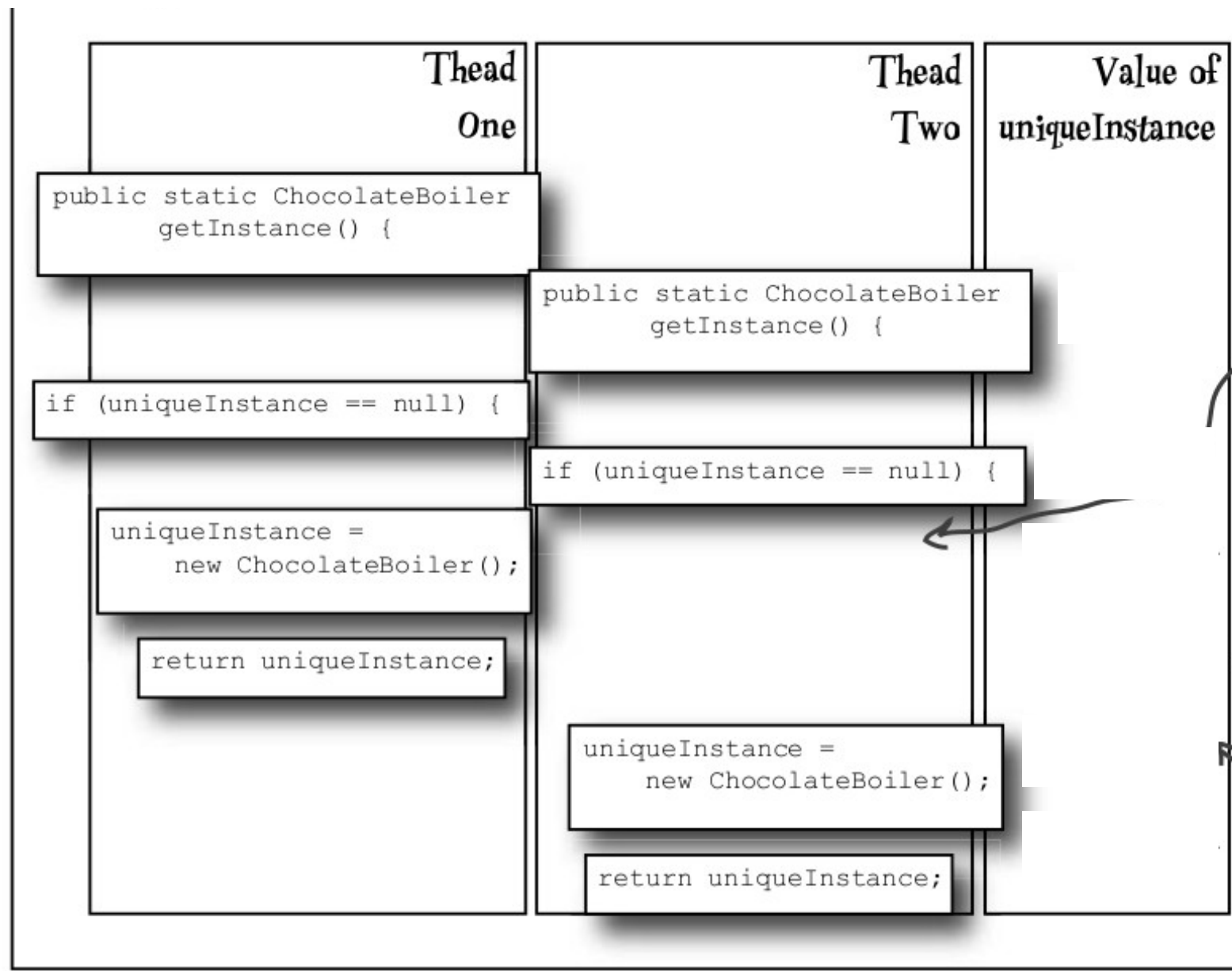


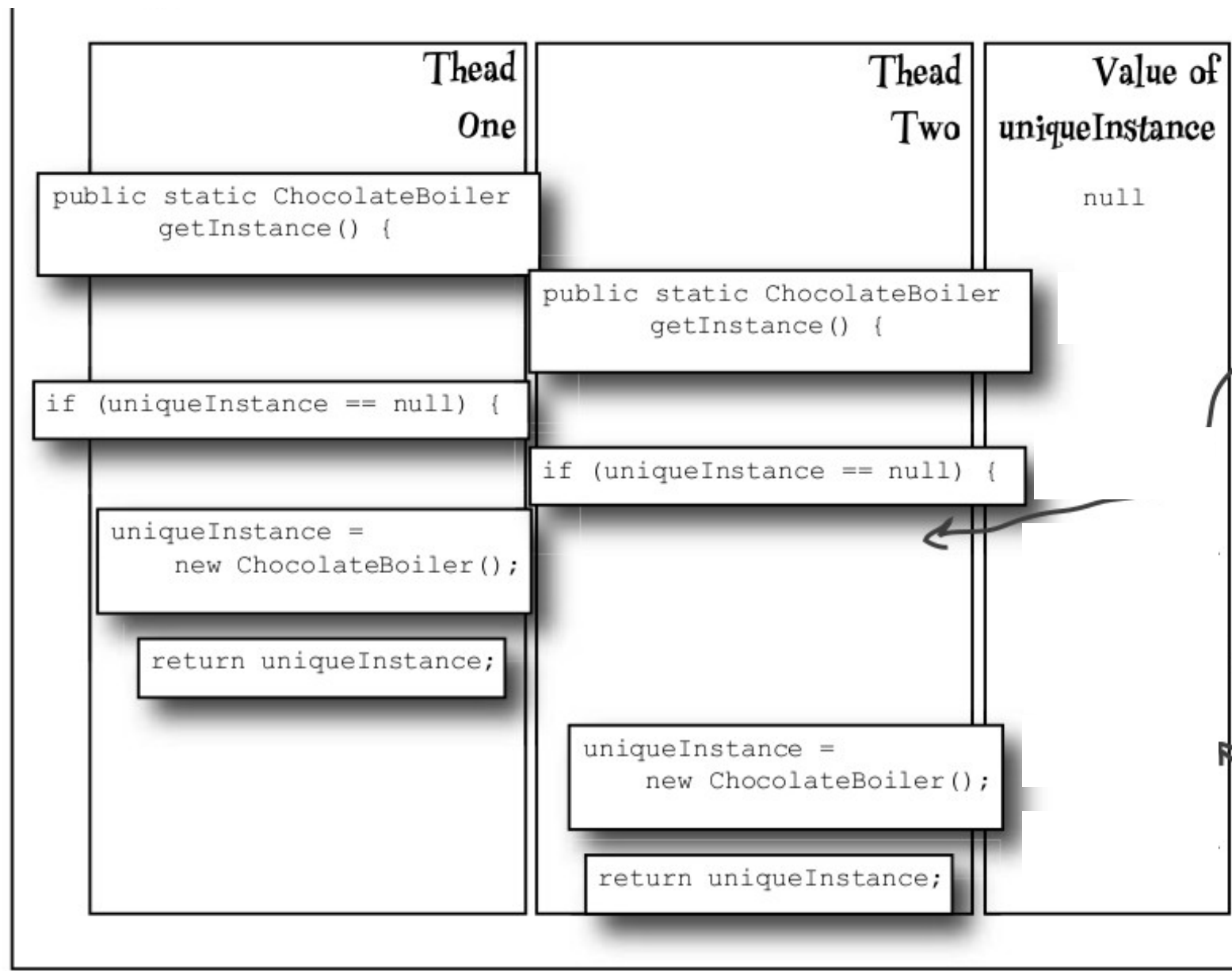
Empresa

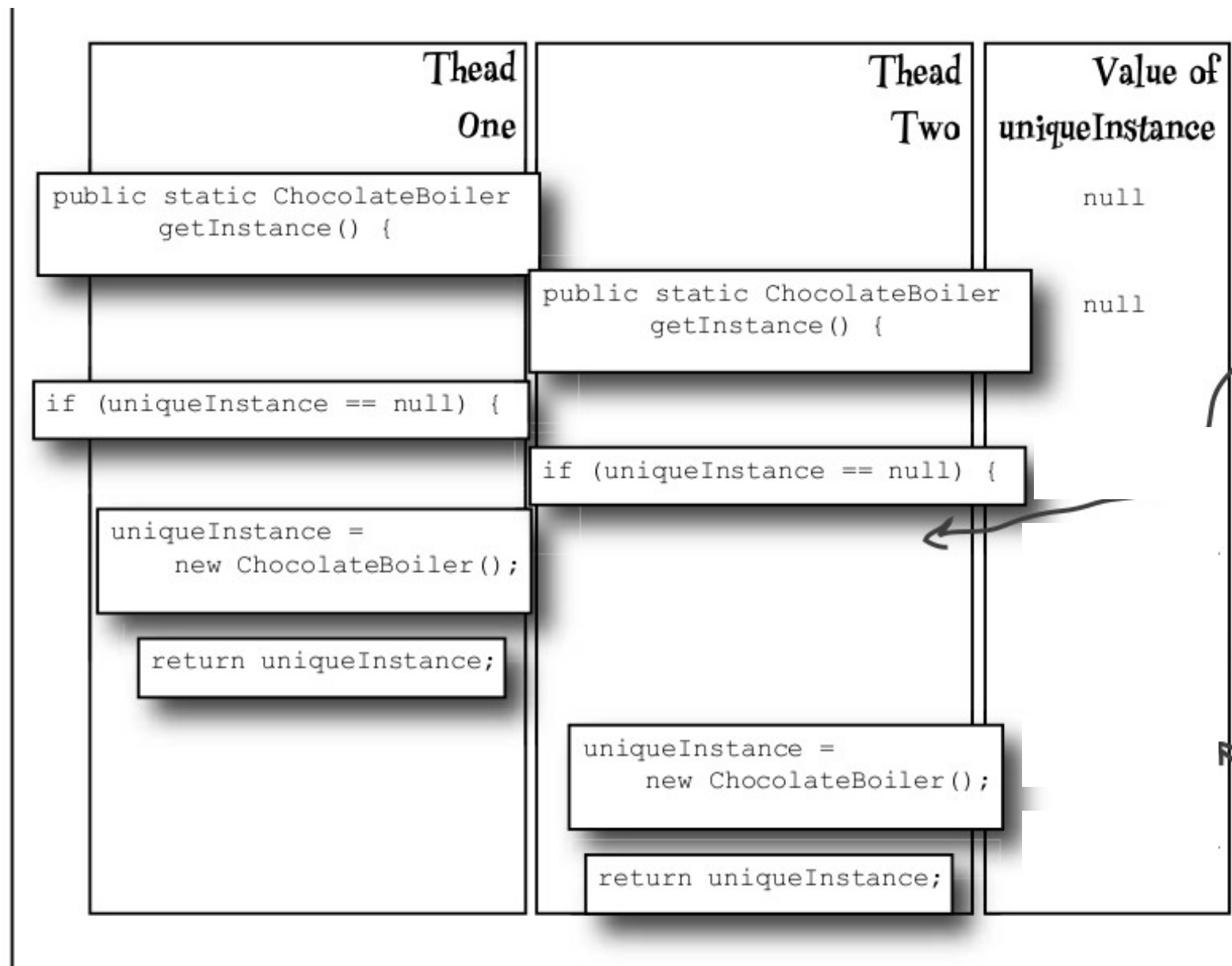
- ¡No sabemos lo que sucedió! El nuevo código de Singleton estaba funcionando bien. Lo único que podemos pensar es que acabamos de agregar algunas optimizaciones al Controlador de la caldera de chocolate que hace uso de múltiples hilos.

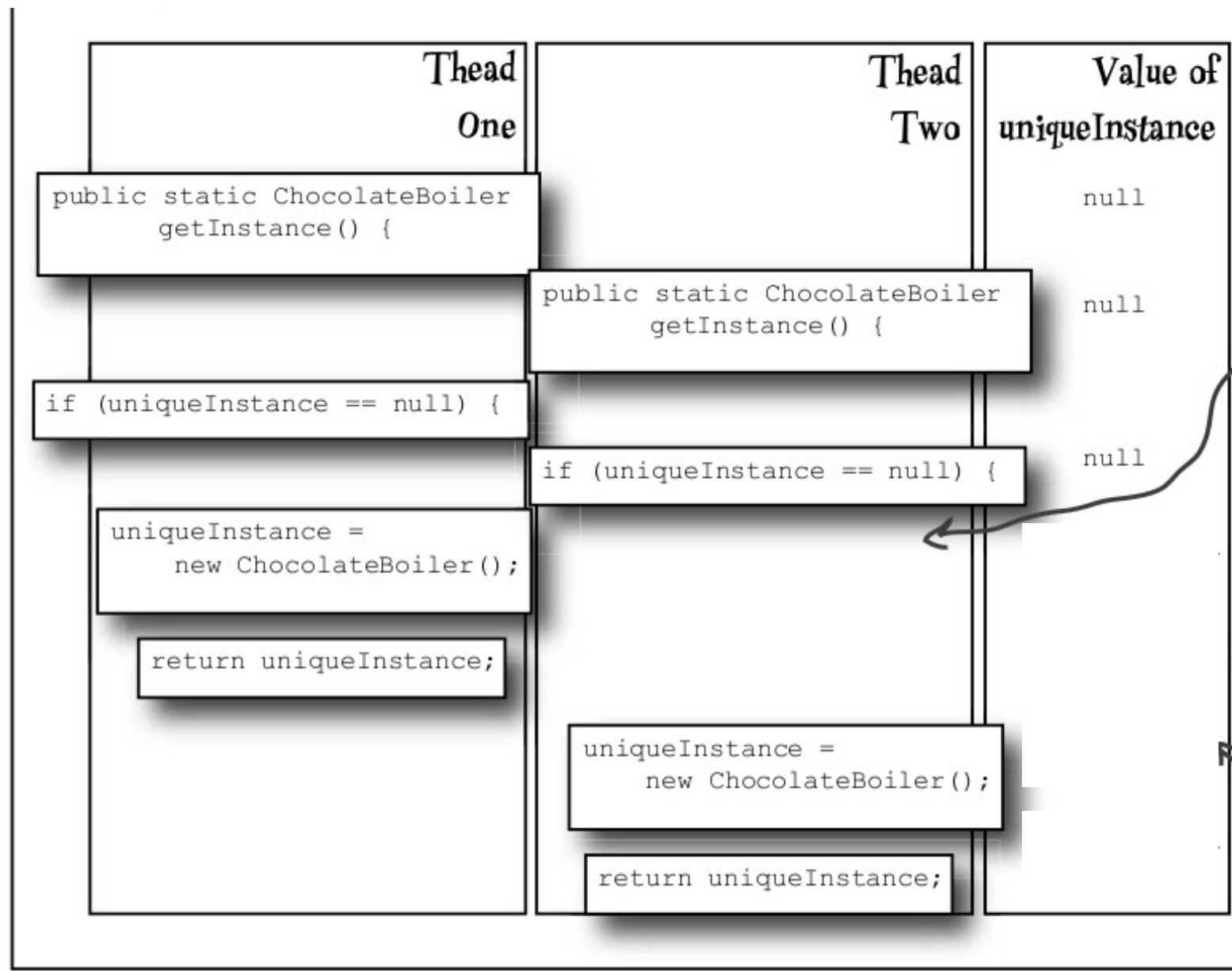
¿Podría la adición de hilos haber causado esto? ¿No es el caso que una vez que hemos establecido la variable `uniqueInstance` en la única instancia de `ChocolateBoiler`, todas las llamadas a `getInstance()` deberían devolver la misma instancia?

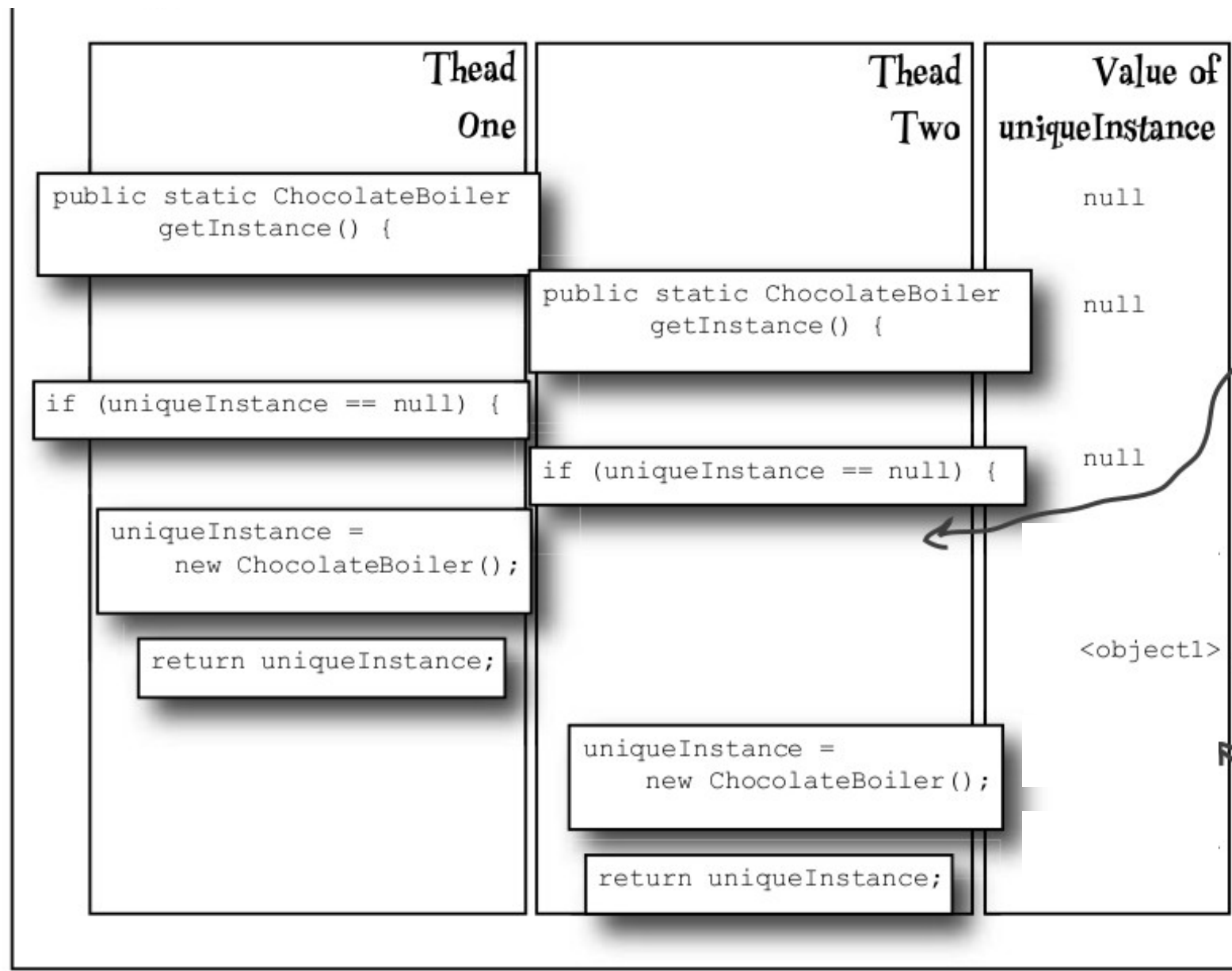
Tenemos dos hilos, cada uno ejecutando este código. Tú trabajo es jugar a la JVM y determinar si hay un caso en el que dos hilos puedan tener acceso a diferentes objetos de la caldera.

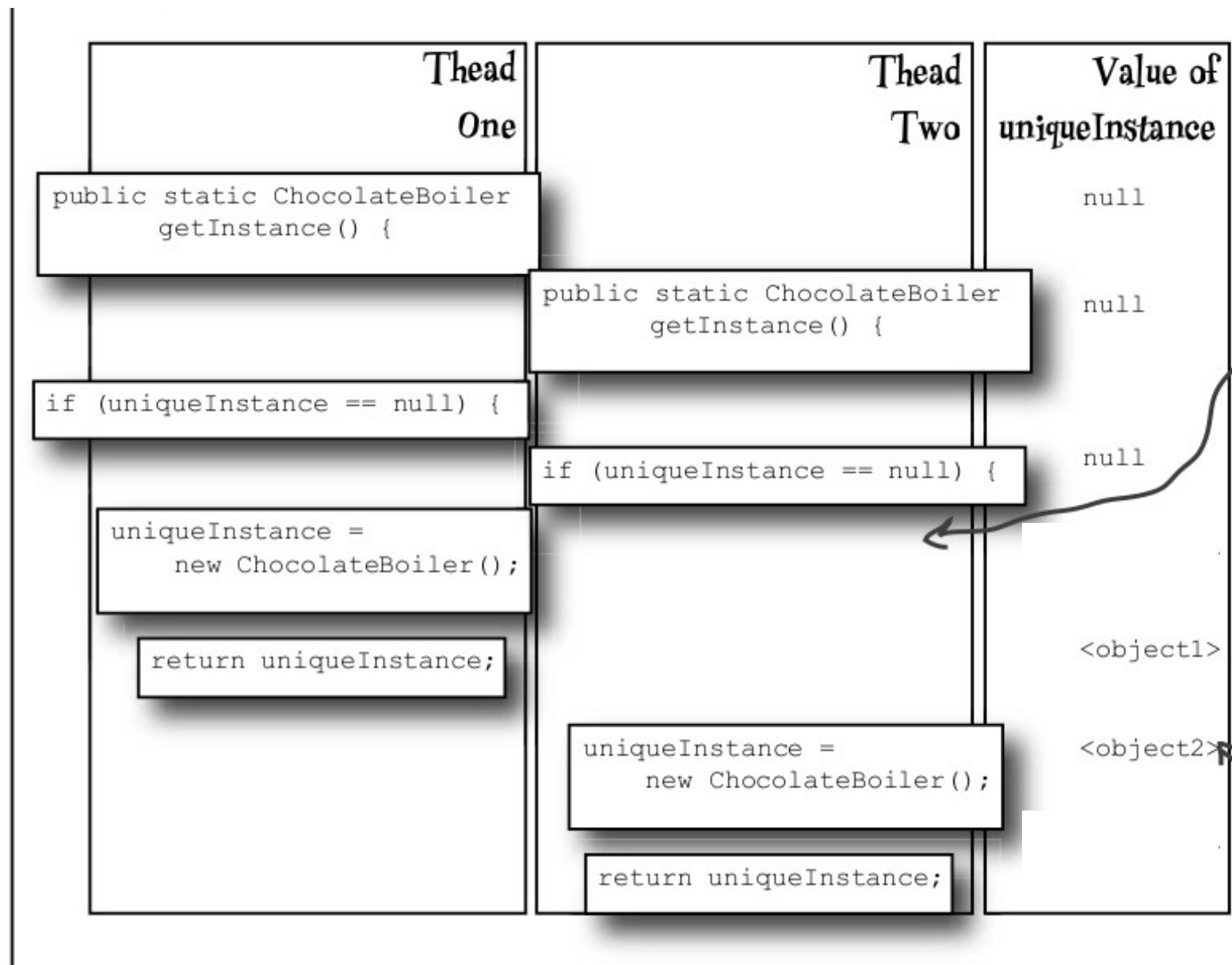


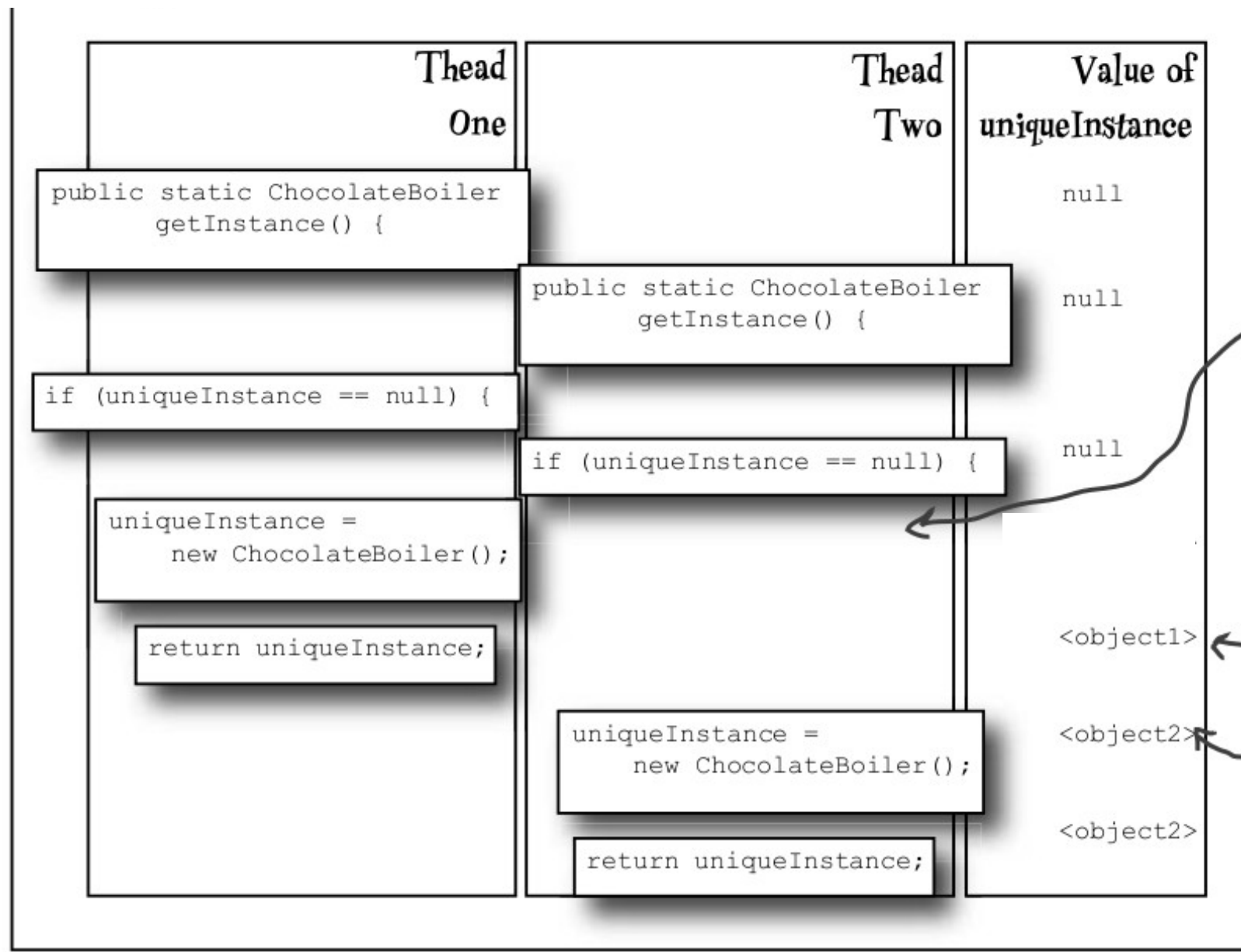












Uh oh, this doesn't look good!

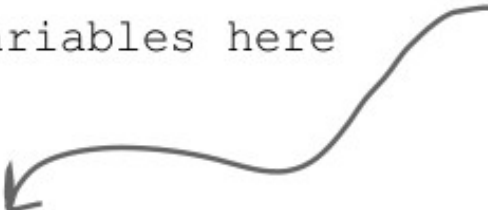
Two different objects are returned! We have two ChocolateBoiler instances!!!

Tratando con multihilo

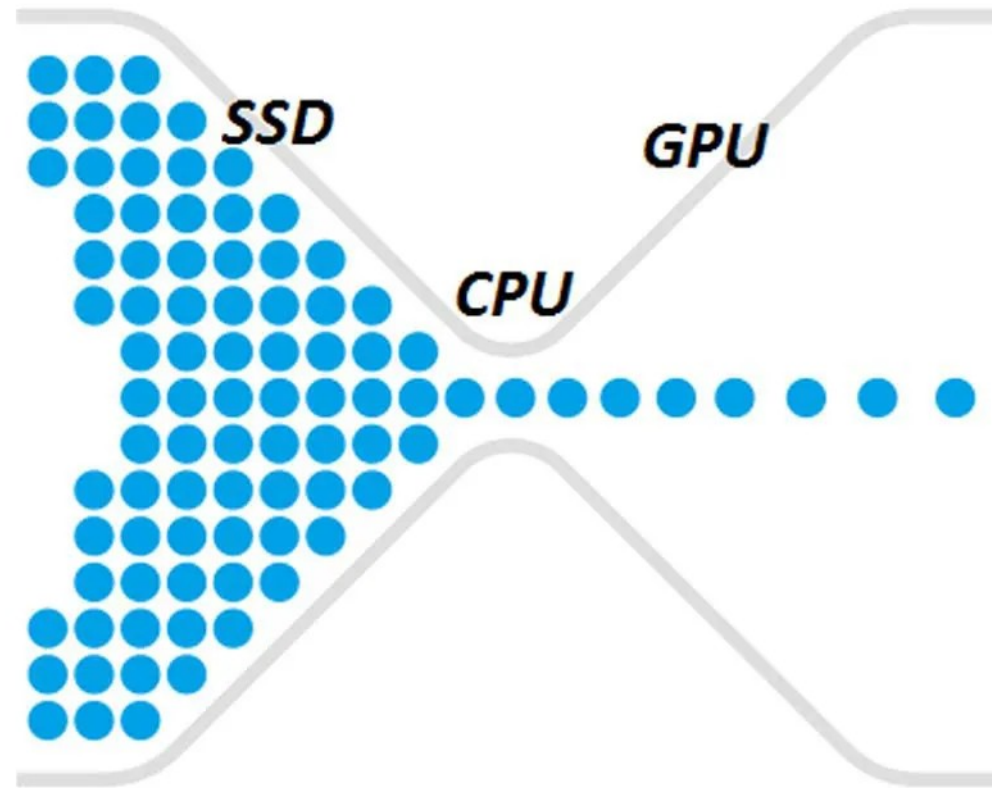
Nuestros problemas de subprocesos múltiples se resuelven casi trivialmente al hacer que `getInstance ()` sea un método sincronizado:

```
public class Singleton {  
    private static Singleton uniqueInstance;  
  
    // other useful instance variables here  
  
    private Singleton() {}  
  
    public static synchronized Singleton getInstance() {  
        if (uniqueInstance == null) {  
            uniqueInstance = new Singleton();  
        }  
        return uniqueInstance;  
    }  
  
    // other useful methods here  
}
```

By adding the `synchronized` keyword to `getInstance()`, we force every thread to wait its turn before it can enter the method. That is, no two threads may enter the method at the same time.



Estoy de acuerdo, esto solucionó el problema. Pero la sincronización es costosa; ¿Es esto un problema?



Estoy de acuerdo, esto solucionó el problema. Pero la sincronización es costosa; ¿Es esto un problema?

- En realidad es un poco peor de lo que crees: el único momento en que la sincronización es relevante es la primera vez que se usa este método. En otras palabras, una vez que establezcamos la variable `uniqueInstance` en una instancia de Singleton, no tenemos necesidad de sincronizar este método. Después de la primera vez, ¡la sincronización es totalmente innecesaria!

¿Podemos mejorar el multihilo?

- Para la mayoría de las aplicaciones Java, obviamente necesitamos asegurarnos de que Singleton funcione en presencia de múltiples hilos. Pero parece bastante caro sincronizar el método `getInstance()`, entonces, ¿qué hacemos? Bueno, tenemos algunas opciones ...

1. No hacer nada, si el rendimiento de getInstance() no es crítico para su aplicación

Está bien; si llamar al método getInstance () no está causando una sobrecarga sustancial para su aplicación, olvídense de ello. La sincronización de getInstance () es directa y efectiva. Solo tenga en cuenta que sincronizar un método puede disminuir el rendimiento en un factor de 100, por lo que si una parte de alto tráfico de su código comienza a usar getInstance (), puede que tenga que reconsiderarlo.

2. Vaya a una instancia procesada en lugar de a una instancia creada

Si su aplicación siempre crea y usa una instancia de Singleton o la sobrecarga de la creación y los aspectos del tiempo de ejecución del Singleton no son pesados, puede crear su Singleton, así:

```
public class Singleton {  
    private static Singleton uniqueInstance = new Singleton();  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        return uniqueInstance;  
    }  
}
```

Go ahead and create an instance of Singleton in a static initializer. This code is guaranteed to be thread safe!

We've already got an instance, so just return it

Usando este enfoque, confiamos en la JVM para crear la instancia única de Singleton cuando se carga la clase. La JVM garantiza que la instancia se creará antes de que cualquier subprocesso tenga acceso a la variable staticInstance estática.

3. Use "bloqueo comprobado doble" para reducir el uso de sincronización en getInstance ()

Con el bloqueo comprobado, primero verificamos si se crea una instancia, y si no, ENTONCES sincronizamos. De esta manera, solo sincronizamos la primera vez, justo lo que queremos.

Vamos a ver el código:



```
public class Singleton {  
    private volatile*static Singleton uniqueInstance;  
  
    private Singleton() {}  
  
    public static Singleton getInstance() {  
        if (uniqueInstance == null) {  
            synchronized (Singleton.class) {  
                if (uniqueInstance == null) {  
                    uniqueInstance = new Singleton();  
                }  
            }  
        }  
        return uniqueInstance;  
    }  
}
```

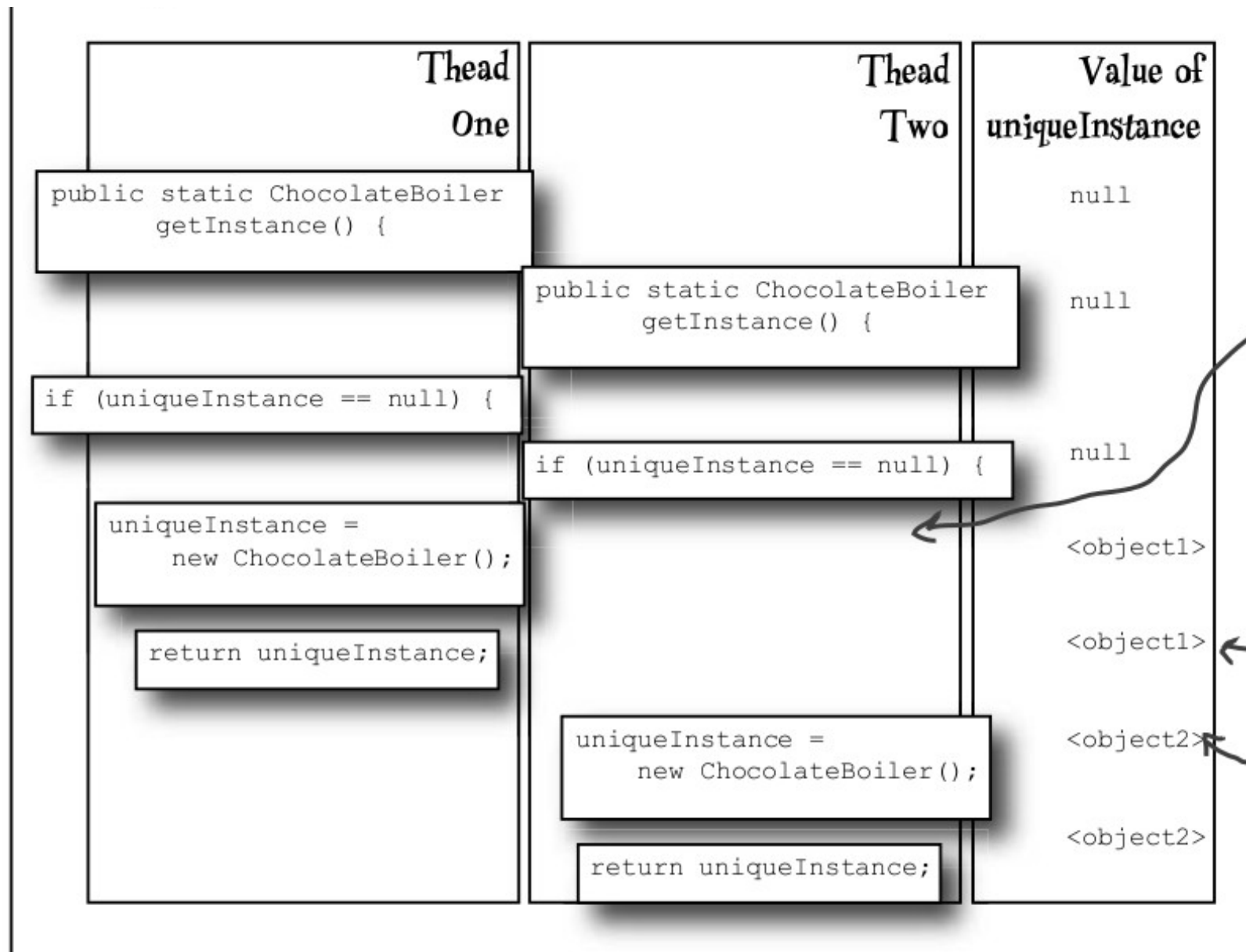
Check for an instance and if there isn't one, enter a synchronized block.

Note we only synchronize the first time through!

Once in the block, check again and if still null, create an instance.

* The volatile keyword ensures that multiple threads handle the uniqueInstance variable correctly when it is being initialized to the Singleton instance.

Si el rendimiento es un problema en el uso del método getInstance(), entonces este método de implementación de Singleton puede reducir drásticamente la sobrecarga.



Uh oh, this doesn't look good!

Two different objects are returned! We have two ChocolateBoiler instances!!!

Problemas y soluciones con este patrón

- X** El patrón no funciona como tal con programas que usen hilos para acceder a la clase que lo usa.
 - ✓ Existen varias maneras de controlar el acceso a los métodos, se puede hacer que la clase sea Synchronized o el método de getInstance() lo sea.
- 