***Imagen que contiene Logotipo

Descripción generada automáticamente***

Alejandro Ortega Martínez

Grado en Ingeniería Informática

23/4/2021

Tema 2

Exclusión Mutua

Programación Concurrente

Contenido

[1. Introducción 2](#_Toc70076973)

[2. Estructuras básicas de soporte a la concurrencia en Java 3](#_Toc70076974)

[2.1. Ciclo de vida de un hilo 3](#_Toc70076975)

[2.2. Threads 3](#_Toc70076976)

[2.3. Prioridades 4](#_Toc70076977)

[2.4. Dormir hilos 4](#_Toc70076978)

[2.5. Esperar a hilos 4](#_Toc70076979)

[2.4. Interrupciones 5](#_Toc70076980)

[2.6. Depósito de hilos (pools) 6](#_Toc70076981)

[2.7. Ejecutores y depósitos 6](#_Toc70076982)

[3. Objetos y concurrencia 7](#_Toc70076983)

[3.1. Concurrencia 7](#_Toc70076984)

[3.2. Estructuras para la programación concurrente 7](#_Toc70076985)

[3.3. Concurrencia y programación OO 7](#_Toc70076986)

[3.4. Transformaciones y modelos de objetos 7](#_Toc70076987)

[Transformaciones secuenciales 8](#_Toc70076988)

[Modelo de objetos activos o modelos de actores 8](#_Toc70076989)

[Modelos mixtos 9](#_Toc70076990)

[4. Imposiciones de diseño 10](#_Toc70076991)

[4**.**1. Seguridad 11](#_Toc70076992)

[4.2. Vivacidad 11](#_Toc70076993)

[4.3. Rendimiento 12](#_Toc70076994)

[4.4. Reutilización 12](#_Toc70076995)

# 1. Exclusión

En un sistema **seguro cada objeto se protege contra las violaciones de integridad.**

Las técnicas de exclusión:

* Preservan las **invariantes del objeto** y evitan los efectos que resultarían de actuar momentáneamente sobre representaciones de **estados incoherentes**
* Evitan que múltiples hilos **modifiquen o actualicen concurrentemente** representaciones de objetos

Hay tres estrategias para garantizar la exclusión:

* **Eliminar la necesidad** de exclusión (parcial o total). Asegurarse que los métodos nunca modifiquen la representación de un objeto, por lo que no habrá estados incoherentes.
* **Asegurar dinámicamente** que solo un hilo pueda acceder al estado del objeto. Protección con cerrojos.
* **Asegurar estructuralmente** que solo un hilo pueda usar un objeto dado. Ocultación o restricción de acceso al mismo

La combinación de estas tres puede mejorar la seguridad, la vivacidad y el rendimiento.

Los programas concurrentes son menos tolerantes a fallos. La responsabilidad recae en los programadores (las estrategias de exclusión y reglas de diseño deben ser explícitas y visibles). También hay que tener precaución al utilizar código no diseñado para ejecutarse en contextos concurrentes (Utilizar Wrappers).

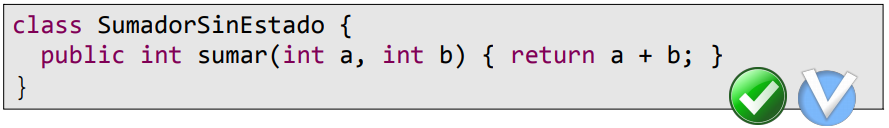
# 2. Inmutabilidad

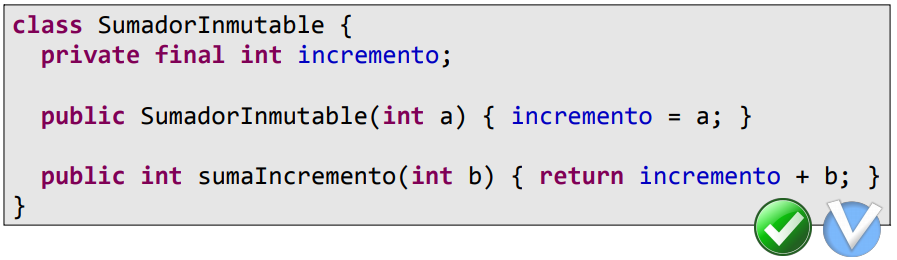
## 2.1. Intención

Si **un objeto nunca puede cambiar de estado**, nunca se encontrará en conflictos o inconsistencias cuando múltiples actividades intenten cambiar de estado de forma incompatible.

**No siempre se puede aplicar**, pero un uso selectivo de la inmutabilidad es una herramienta básica de la programación concurrente.

Podemos conseguir un objeto inmutable si el objeto no tiene ningún campo interno, o mediante clases cuyos campos sean todos final.

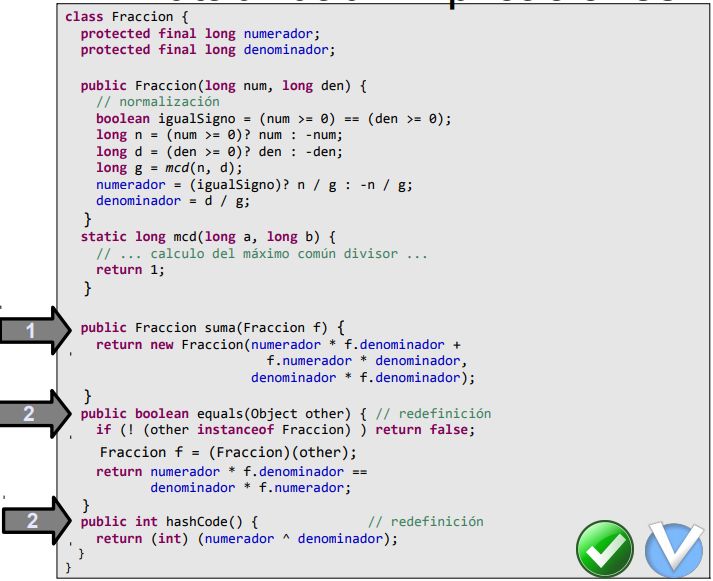




Las instancias de clases inmutables no pueden tener conflictos de Lectura/Escritura o Escritura/Escritura, y no pueden experimentar fallos de invariantes.

## 2.2. Aplicaciones

Hay tipos que se pueden diseñar como inmutables:

* Las instancias de las clases nunca alteran los valores de sus campos pero pueden proporcionar métodos que crean objetos para representar nuevos valores.
* La identidad de los objetos inmutables no importa (Redefinir equals() y hashCode())

**Tipos abstractos de datos** (TDAs): No siempre hay que hacer solo representaciones inmutables.

**Contenedores de valores**: Cuando se necesita establecer un estado coherente y depender de el para siempre.

## 2.3. construcción

Uso apropiado de la **palabra clave final**. Precaución al inicializar objetos inmutables.



En cuanto a los **constructores:**

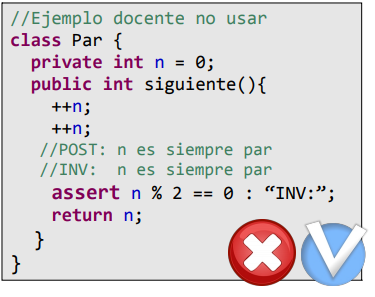
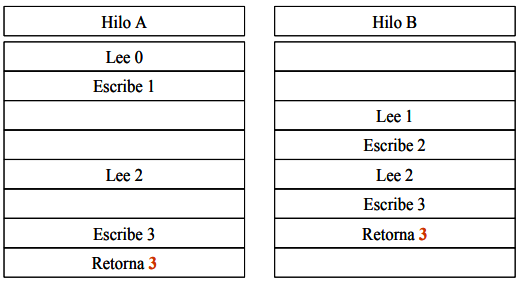
* No deben invocar ningún método cuyos efectos dependan del objeto que está siendo construido.
* Deben evitar almacenar una referencia a un objeto que se haya construido con campos o tablas accesibles por otros, evitando hacer llamadas con this como argumento.

Si la construcción del objeto se inicializa incrementalmente desde archivo o interdependencias entre distintos objetos que se construyan al mismo tiempo, suele implicar **sincronización**.

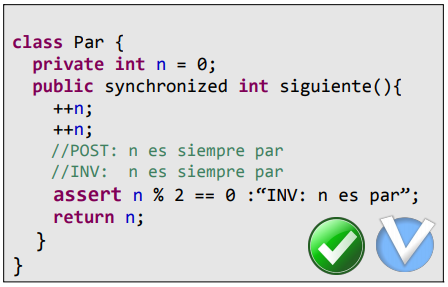
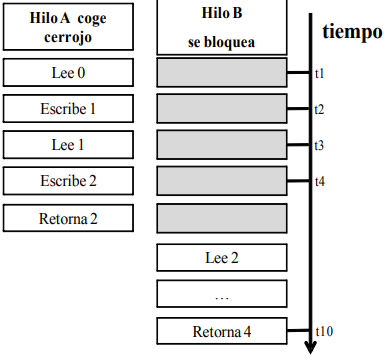
# 3. Sincronización

Los **bloqueos** nos protegen contra conflictos de almacenamiento de bajo nivel y los correspondientes fallos de invariantes de alto nivel.

Violación de la **postcondición**: Suelen ser raras y difíciles de comprobar pero pueden tener efectos devastadores.



Si se declara el método siguiente como **syncronized** se evitan los conflictos. El bloqueo provoca la **ejecución en serie** de los métodos sincronizados.



## 3.1. Formas de conseguir la sincronización

**Objetos y cerrojos**

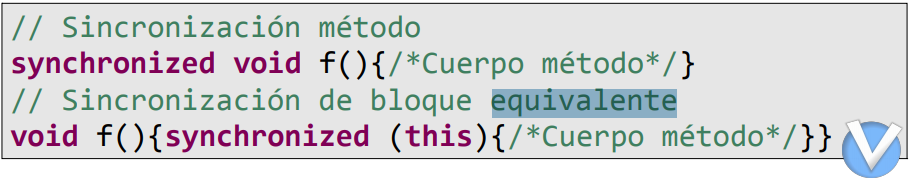
Cada instancia de la clase Object y de sus subclases poseen un cerrojo. Los **campos individuales** no se pueden marcar con syncronized. El bloque solo se puede aplicar al **uso de** **campos dentro ne métodos**. El bloqueo de un array de Objects no bloquea sus elementos. Los cerrojos asociados con métodos con alcance de clase utilizan syncronized static.

**Métodos y bloques sincronizados**

Son dos construcciones sintácticas basada en la palabra clave “synchronized”

Bloques

Toman como argumento el objeto que bloquea. Un argumento común puede ser “this”



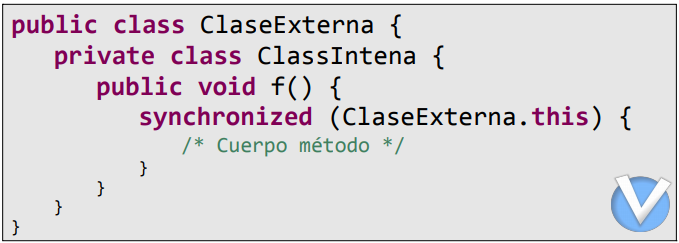
Métodos sincronizados

“Synchronized” no forma parte de la signatura del método.

**No se hereda la sincronización** (Las interfaces no pueden tener métodos synchronized). Los métodos de instancia de la subclase emplean el mismo cerrojo que su superclase.

Los **constructores no pueden ser synchronized**, aunque si que pueden contener bloques synchronized.

La sincronización de un **método de clase interna** es independiente de su clase externa. Un método interno no estático de la clase interna puede bloquear la clase que la contiene mediante el uso de bloques de código.



**Adquisición y liberación de cerrojos**

Es un protocolo definido con la palabra synchronized. Un cerrojo:

* Se **adquiere** a la entrada del método o bloque synchronized.
* Se **libera** a la salida, incluso si se termina con el lanzamiento de una execpcion.

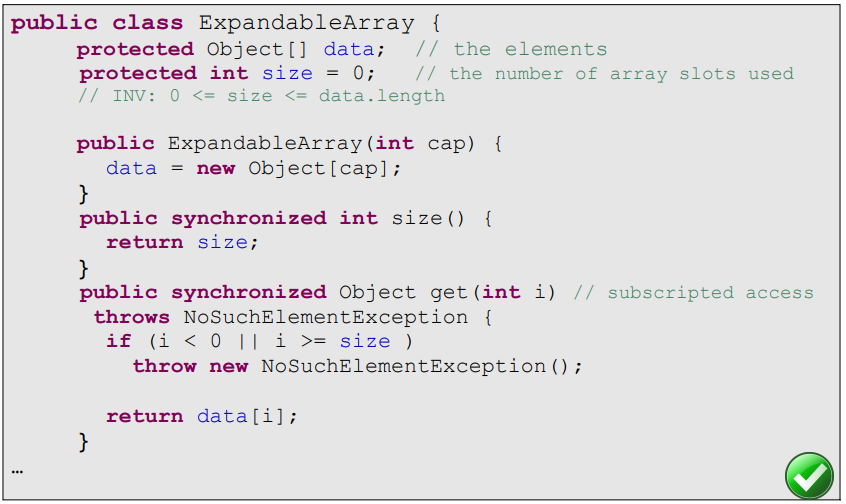
Los cerrojos funcionan en el contexto de hilos y no en la invocación de métodos: Un hilo **avanza** por un método synchronized si el cerrojo **está libre**, o ya lo posee. En otro caso el hilo se bloquea. Los métodos no synchronized pueden ejecutarse en cualquier momento incluso si un método synchronized se está ejecutando.

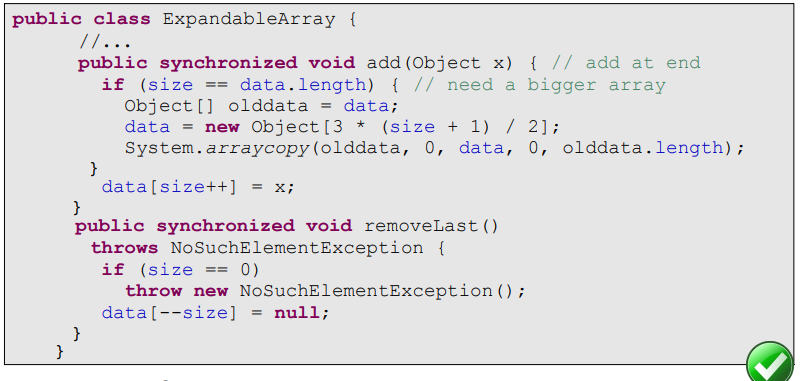
Cuando un hilo libera un cerrojo, otro puede adquirirlo, pero no podemos saber cual de los hilos bloqueados lo adquirirá. Tampoco hay un método para saber si un cerrojo ha sido adquirido.

El bloqueo de un objeto **no protege automáticamente los campos static**. Esto se hace mediante synchronized static.

## 3.2. Objetos completamente sincronizados

Los cerrojos se pueden utilizar para **bloquear a los clientes** que intenten invocar un método de un objeto mientras que otro método o bloque de código esta ejecutando dicho método.

La estrategia mas segura de diseño concurrente es utilizar objetos **completamente sincronizaos o atómicos**. Todos los metodos son synchronized y finitos, no hay campos publicos ni violaciones de encapsulacion, todos los campos se inicializan a un estado consistente en los constructores, y el estado del objeto es consistente siempre al principio y fin de cada metodo.



No hay conflictos Lectura/Escritura (get() en mitad de un removeLast()), y se eliminan los conflictos Escritura/Escritura (dos add() concurrentes).

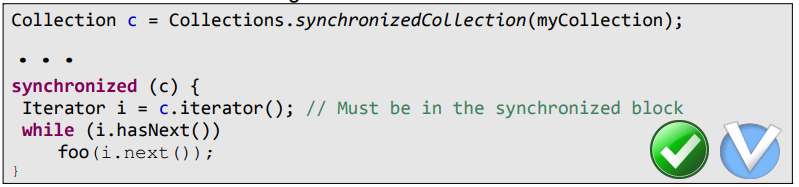
## 3.3. Estrategias de recorridos en colecciones

**1. Con bloqueo en el lado del cliente**

El api java.util.Collection utiliza un **wrapper** para obtener una colección completamente sincronizada. Añade métodos de envoltura del tipo:



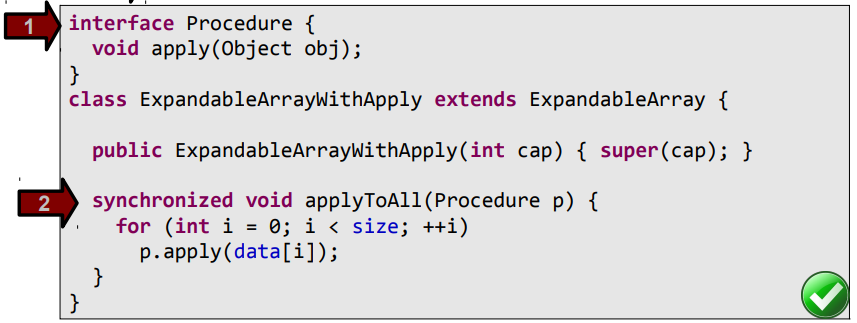
Esto devuelve una colección sincronizada a partir de la colección especificada, con el fin de garantizar el acceso secuencial. Sin embargo, el **usuario es el encargado de sincronizar los recorridos**:



**2. Operaciones sincronizadas agregadas en la colección**

Se puede asegurar el recorrido si:

* Se abstrae la función que se quiere aplicar a cada elemento de la colección.
* Se define un método sincronizado en la colección.



**3. Iteradores con control de versiones**

Lanzan una excepción si la colección se modifica a mitad de recorrido. Una forma de implementarlo:

* Mantener un numero de versión que se incrementa en cada actualización de la colección.
* El iterador puede comprobar ese valor siempre que se pida el elemento siguiente y lanza una excepción si ha cambiado

Esta estrategia se emplea en java.util.Iterator, que lanza la excepción ConcurrentModificationExcepcion.

## 2.4. Singleton

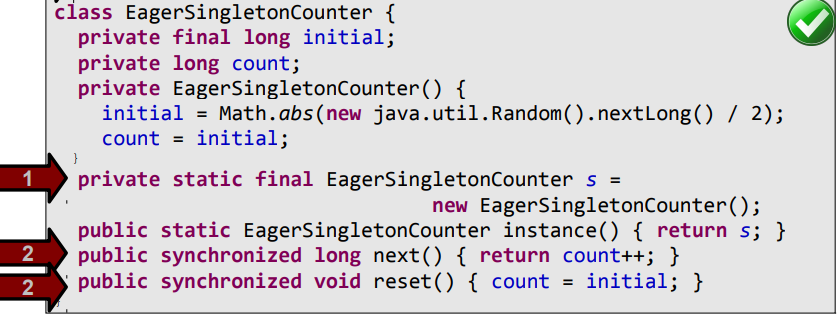
Su intención es **garantizar que una clase solo tenga una instancia**, y proporciona un punto de acceso global a ella.

**Solución no concurrente:** La clase Singleton define una variable estática que referencia a la única instancia de la clase. Para prevenir que los clientes creen mas instancias se declara el constructor privado. Para permitir el acceso a la instancia, la clase proporciona un **metodo estático** (getInstance()).

En **concurrencia** dos hilos distintos puedeninvocar almetodo getInstance() en el mismo instante de tiempo.

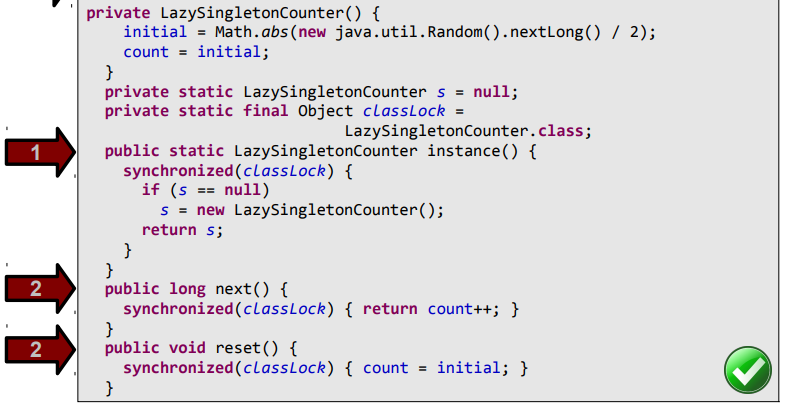
**Solución concurrente 1**

* Evitar inicialización retardada.
* Sincronizar completamente el objeto

****

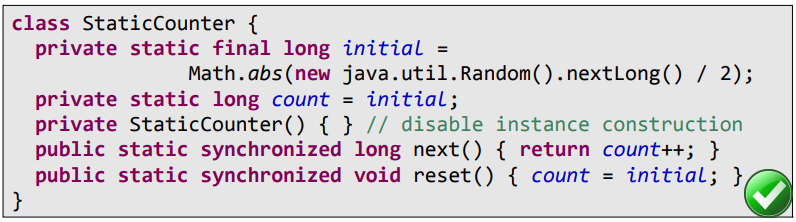
**Solución concurrente 2**

* Bloqueo de clase en la inicialización de la instancia
* Bloqueos de clase en el cuerpo de los métodos



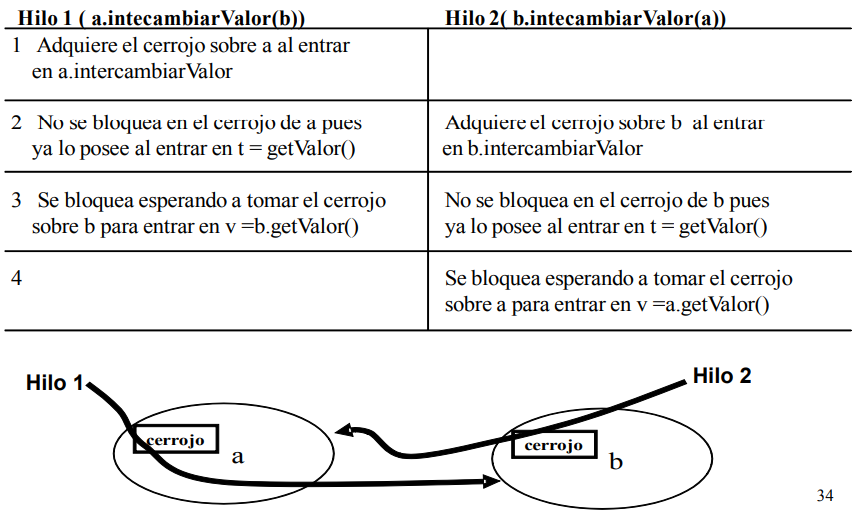
**Solución concurrente 3**

Si no es necesario depender de las instancias, utilizar todos los métodos estáticos sincronizados. No hacen falta constructores.



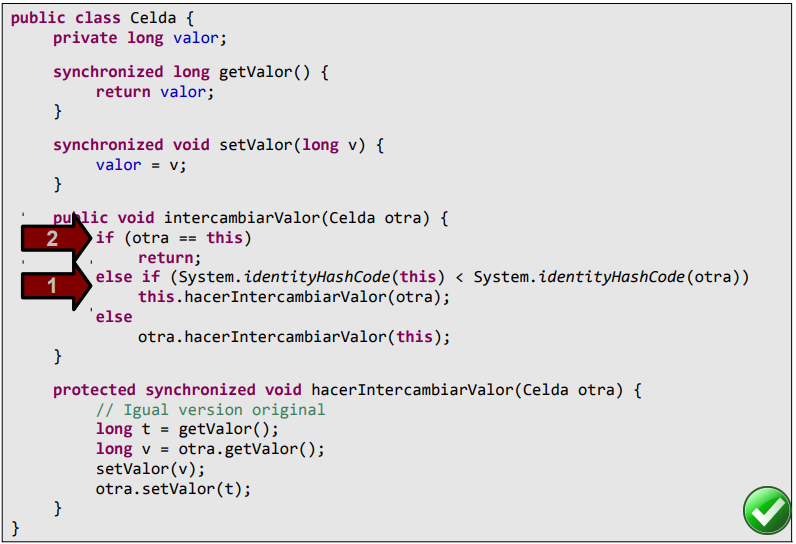
## 3.5. Interbloqueo

Seguridad vs Vivacidad. Los objetos atómicos son siempre seguros, pero los hilos que lo usan no están siempre activos.



**Ordenación de recursos**

Es una técnica sencilla para evitar el interbloqueo.

Se asocia una etiqueta numérica a cada objeto que se pueda bloquear en un bloque o método synchronized.

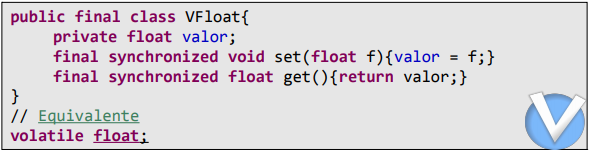
* Si la sincronización siempre se realiza del de menor orden con respecto a las etiquetas nunca se da el interbloqueo.
* Se debe comprobar la **detección del alias** (Dos o más referencias se asignan al mismo objeto)

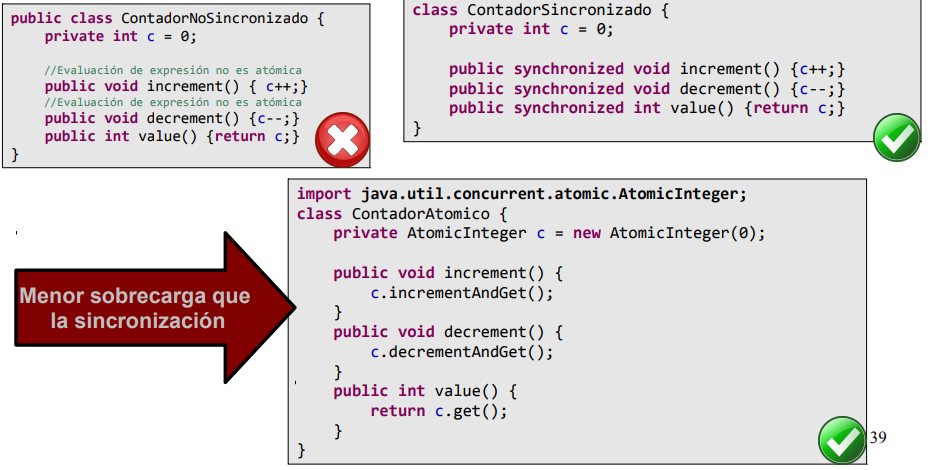
## 3.6. Atomicidad



El modelo de memoria en java permite lecturas y escrituras atómicas de celdas de memoria. Cualquier tipo primitivo excepto (long o double) tiene garantizada la atomicidad.

**Volatile** extiende la atomicidad a referencias a objetos y a float y double. La atomicidad **no elimina la necesidad de sincronizar acciones atómicas**.





# 4. Confinamiento

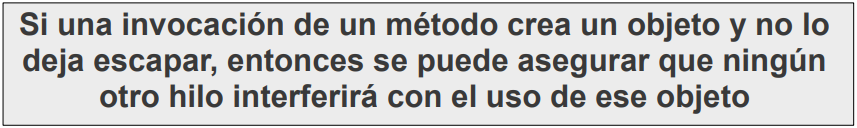
Consiste en emplear técnicas de encapsulamiento para **garantizar estructuralmente** que como mucho, una actividad puede acceder un objeto determinado (elimina la necesidad de bloqueos dinámicos)

La táctica es definir métodos y clases que establezcan **dominios de propiedad herméticos** que garantizan que un hilo o un hilo a la vez puede tener acceso a un **objeto determinado**.

El confinamiento depende del ámbito, del control de acceso y las características del lenguaje respecto a la ocultación de datos y encapsulamiento.

## 4.1. Confinamiento de metodos

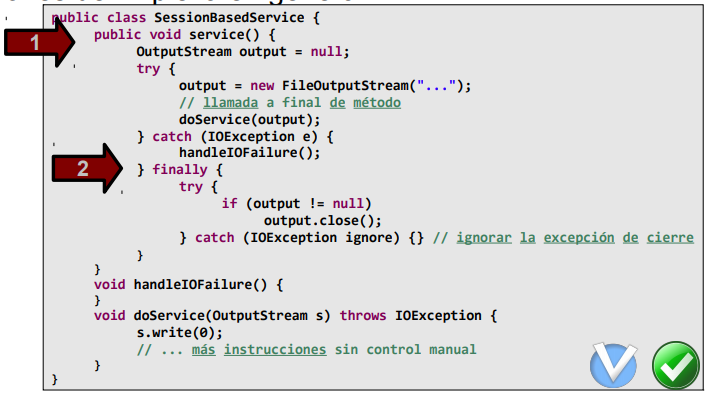
Implica práctica de programación ordinaria que tienen que ver con variables locales.



Es una ocultación dentro de ámbitos locales, y esta técnica se puede extender en las secuencias de invocaciones del método.

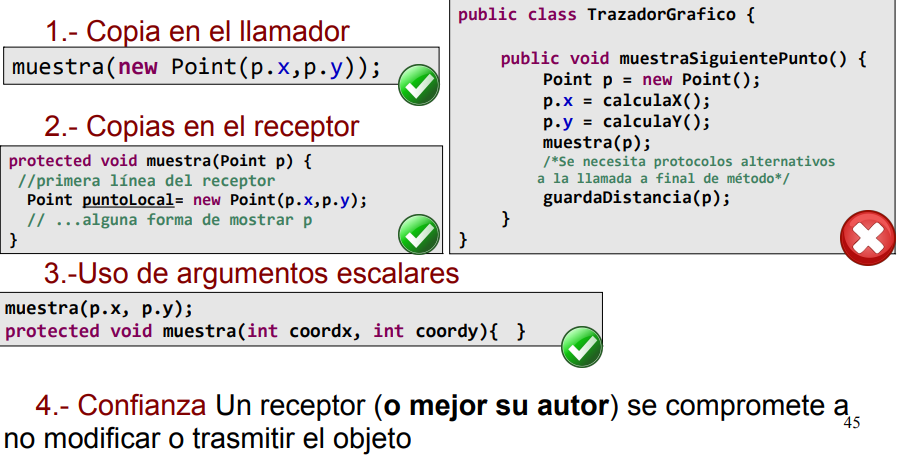
La llamada a final del metodo es un protocolo sin intervención explicita, que asegura que en un momento dado como máximo puede tener acceso a un objeto un único metodo que se ejecuta activamente.

**Sesiones**

* Un cierto metodo público de entrada construye los objetos que serán confinados en una secuencia de operaciones que comprenden un servicio
* El metodo de entrada también es responsable de las operaciones de limpieza en general.

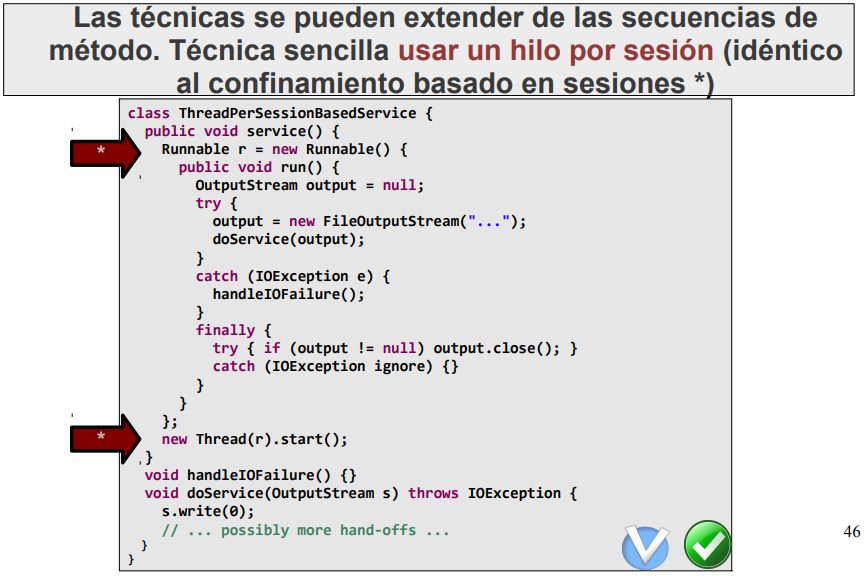
**Protocolos alternativos**

Si un metodo debe tener acceso a un objeto despues de una llamada, o debe hacer llamadas múltiples no puede aplicarse el protocolo llamada a final de metodo.



## 4.2. Confinamiento dentro de hilos

Se trata de técnicas para restringir el acceso dentro de hilos.

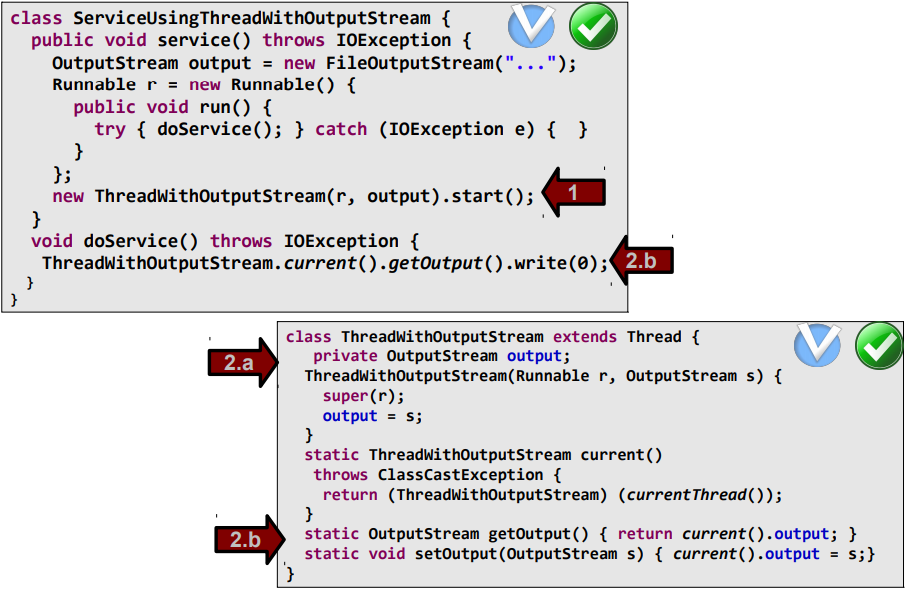


Alternativas de diseño concurrente para el **confinamiento de datos dentro de hilos:**

1. Requerir que todos los campos que son accesibles dentro del hilo entren **estrictamente** confinados en ese hilo
2. En JVM se comparten accesos de los recursos subyacentes

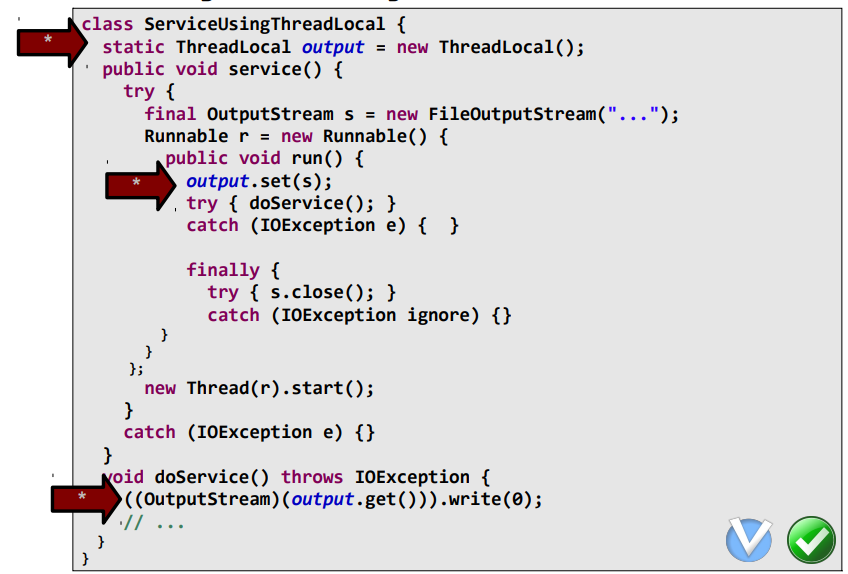
Como confinar campos específicos dentro de hilos:

1. Recibir referencias confinadas a lo largo de secuencias de llamadas
2. Extender de la clase Thread
   1. Añadir campos específicos
   2. Añadir métodos para acceder a los campos del hilo actual



Java.lang.ThreadLocal permite que los datos específicos del hilo sean accesibles desde cualquier código de manera concurrntemente apropiada:

* Mantiene una tabla que asocia los datos específicos a instancias del Thread.ThreadLocal
* Proporciona los metodos set y get para tener acceso a los datos que posee el Thread actual

Es una alternativa que elimina la dependencia de extencion sobre la clase Thread.

## 4.2. Confinamiento de objetos

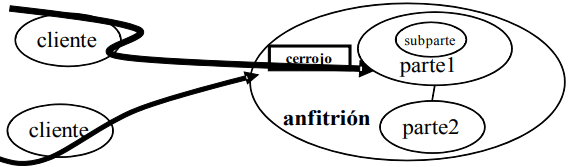
Es una técnica de encapsulamiento OO para asegurar la unicidad de acceso para los metodos que incorporan objetos.

Cuando no se puede restringir el acceso a un **objeto anfitrión** dentro de un metodo o de un hilo:

* Se debe utilizar bloqueo dinámico
* Restringir todos los accesos internos a ese objeto (Elimina la necesidad de bloqueos adicionales)
* El control de la exclusión para el objeto anfitrión se propaga automáticamente a todas sus partes internas (Asegura la sincronización en todos los puntos de entrada del objeto anfitrion) (Objetos completamente sincronizados)

El objeto anfitrión posee partes internas y **es responsable** de:

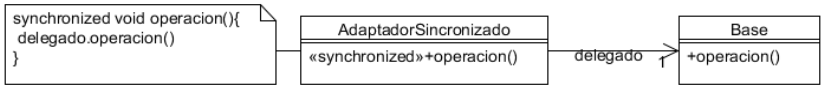
* Construir nuevas instancias de cada objeto parte asignando referencia a campos privados.
* Nunca debe perder las referencias de los objetos parte.
* Debe tener sus métodos sincronizados.
* En la variante más conservadora (**contención fija**), el objeto anfitrión nunca reasigna los campos de referencia que apuntan a objetos parte internos.

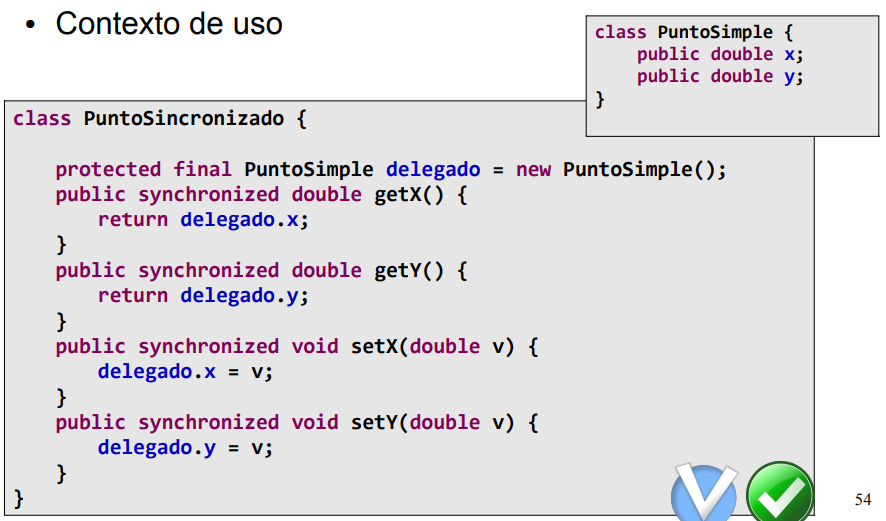


**Adaptadores**

Su intención en un contexto concurrente es conectar **objetos base** no sincronizados dentro de objetos **anfitriones completamente sincronizados**. El anftrión/adaptados proporciona el punto de entrada seguro.

Dada una **clase base no protegida**, se puede definir una **clase adaptador sincronizada** con un campo, llamado **delegado**, que contiene una referencia al objeto base, el cual envia peticiones. El acceso al delegado debe ser exclusivo.



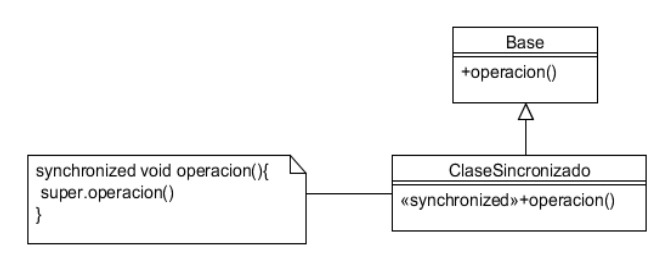


**Creación de subclases**

Es una alternativa a la delegación.

Cuando se tiene la intención de que las instancias de una clase se confinen dentro de otras no hay razón para sincronizar sus metodos.

Cuando se confinan unas si y otras no, lo mas seguro es sincronizar sus metodos.

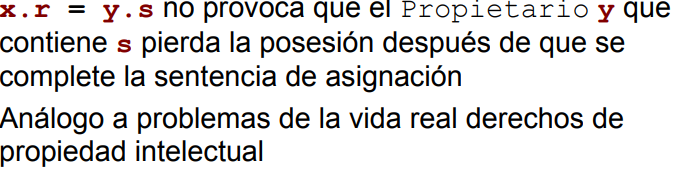


## 4.4. Confinamiento de grupos

Amplia las técnicas a los conjuntos de colaboración de objetos a través de múltiples hilos.

Los protocolos de transferencia del **recurso** son sencillos (Adquirir recurso, dar recurso, intercambiar recurso…)

Los campos que mantienen referencias a objetos no actúan como referencias físicas:



Los **protocolos se pueden encapsular en los metodos** realizando las distintas operaciones para distintos objetos Recursos r y s, y los objetos Propietarios x e y que pueden almacenarlos en un campo ref.

(Se muestran los cerrojos con bloques sinchronized)

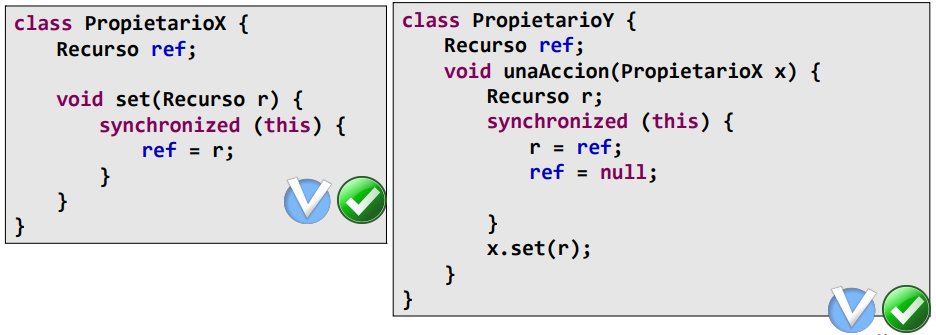
* **Adquirir** el Propietario x se apropia inicialmente de r. Generalmente es el resultado de construir o inicializar r de la siguiente forma:



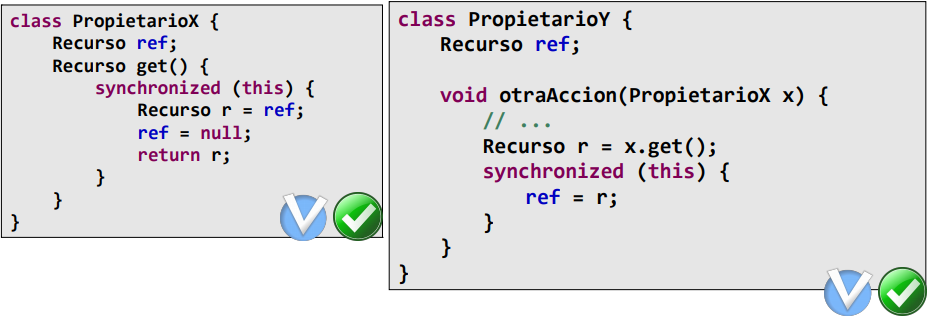
* **Olvidar** el Propietario x hace que el uso de r no sea poseido por ningún propietario. Esto lo consigue el propietario actual realizando:



* **Dar (set)** el propietario y envia un mensaje al propietario x que contiene una referencia al Recurso r como argumento, despues de que y deje de tener posesión de r y se le ceda a x.



* **Coger (get)** el propietario y solicita el Recurso r al propietario x que envía r como valor de retorno abandonando la posesión.



* **Intercambiar** el propietario y cambia su recurso s por el recurso r al propietario x. Esta operación se puede usar para coger un recurso por medio de s=intercambiar(null), o para cederlo por medio de intercambiar(r), ignorando el resultado.

