THREAD SAFE(SEGURIDAD EN HILOS)

Una pieza de código es segura o "thread safe" en un contexto de ejecución de hilos si funciona correctamente durante la ejecución simultánea de múltiples hilos. Tiene que ser seguro tanto el código de los hilos como el de los objetos que usan los hilos(si un hilo llama al método b() de la clase A, b() también tiene que ser thread safe)
Es un tema muy extenso y complejo y sólo veremos unas pinceladas. Para empezar los dos problemas de seguridad genéricos que ocurren son:

- inconsistencias en los datos compartidos (variable, array, fichero, bd,…)
- interbloqueos entre hilos (se cuelga la aplicación)

Mecanismos java para conseguir código Thread Safe:

- código sincronizado(Synchronized)
- código inmutable (final)
- candados
- semáforos
- comunicación de hilos con con wait() y notify()
- etc.

Hay una variedad de problemas con concurrencia y consecuentemente una variedad de recursos en Java para resolverlos. Veremos un ejemplo de código con Synchronized

Synchronized para proteger secciones críticas de código

Cuando varios hilos pueden compartir datos, las zonas de código que manipulan dichos datos pueden generar resultados inconsistentes. Estas zonas de código se les llama secciones críticas.

Con la palabra Synchronized, se puede indicar que un trozo de código o un método completo es una sección crítica. (nosotros usaremos en este boletín sólo métodos synchronized completos) y esto hace que dicho método se ejecute con *exclusión mútua* de forma que si un thread quiere usar este código, ningún otro thread puede hacerlo hasta que esté libre, una vez libre, puede pasar a usarlo él marcándolo como ocupado para evitar que otros thread intenten utilizarlo. Por lo tanto se trata de "suspender temporalmente" la ejecución concurrente forzando una ejecución secuencial para preservar la integridad de la información.

En el siguiente ejemplo observamos como ciertamente el efecto de synchronized es suspender temporalmente la ejecución concurrente. Tenemos dos hilos que utilizan el código del método sum(). Demostramos que al declararlo synchronized el primer hilo que lo consigue lo bloquea y no lo suelta hasta que acaba su trabajo. Si ejecutas muchas veces este código observa que aunque normalmente el "Thread A" suele ser el primero en bloquear a sum() a veces es el "Thread B" y siempre observamos que este método jamás se ejecuta concurrentemente pues no se mezclan sus print()

```
class A {
   synchronized void sum(int n){
      Thread t = Thread.currentThread();
      for (int i = 1; i <= 5; i++) {
         System.out.println(
            t.getName() + " : " + (n + i));
      }
   }
}


class B implements Runnable {

   A a = new A();
```

```
    public void run(){
        a.sum(10);
    }
}
class App{
    public static void main(String[] args){

        B b = new B();

        Thread t1 = new Thread(b);
        Thread t2 = new Thread(b);

        t1.setName("Thread A");
        t2.setName("Thread B");

        t1.start();
        t2.start();
    }
}
```

Ejercicio U5_B9B_E1: Una cuenta bancaria tiene dos usuarios Juan y Lola. Concurrentemente Juan y Lola sacan dinero, por ejemplo desde cajeros automáticos. Juan ve saldo 100 y quiere sacar 60. Simultáneamente, Lola también ve 100 y saca 50. Ambas operaciones serán autorizadas pues no superan el saldo 100, pero al ocurrir simultáneamente la cuenta queda con -10, es decir, en un estado inconsistente si suponemos que el banco no permite saldos negativos.

Observa el siguiente código con estas consideraciones

- Para evitar que Juan y Lola metan la cantidad a retirar por teclado, hacemos que ambos quiten 60 directamente desde el código. La entrada por teclado con varios hilos en la misma consola es caótica y la evitamos.
- La clase GestorCuenta podrá tener muchos métodos y unos podrán ejecutarse concurrentemente y otros no. Para simplificar supongamos que tenemos sólo un método retirardinero() y que será además problemático ejecutarlos concurrentemente.

```
@Override
    public void run() {
        this.retirarDinero(60);
    }
```

queríamos que este método fuera eficiente y por tanto que se ejecute concurrentemente pero con la ejecución concurrente surge el siguiente problema: se puede quedar la cuenta con saldo negativo

SE PIDE: Solucionar el problema comentado en el siguiente código

```
class CuentaBanco {

 private int saldo = 100;

 public int getSaldo() {
    return saldo;
 }

 public void reintegro(int cantidad) {
    saldo = saldo - cantidad;
 }

}

class GestorCuenta implements Runnable {

 private CuentaBanco cb = new CuentaBanco();
 private void gestionConcurrente(){
    // aquí iría las funciones del gestor, solo ejemplificamos la de retirarDinero()
    retirarDinero(60);
 }
 private void retirarDinero(int cantidad) {
```

```
    if (cb.getSaldo() >= cantidad) {
        System.out
                .println(Thread.currentThread().getName() + " está realizando un Reintegro de: " + cantidad + ".");

        // tiempo de espera para simular que el usuario lee o piensa
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {

            e.printStackTrace();
        }
        cb.reintegro(cantidad);

        System.out.println(Thread.currentThread().getName() + ": Reintegro realizado.");
        System.out.println(Thread.currentThread().getName() + ": Los fondos son de  " + cb.getSaldo());

    } else {
        System.out.println("No hay suficiente dinero en la cuenta para realizar el Reintegro Sr."
                + Thread.currentThread().getName());
        System.out.println("Su saldo actual es de " + cb.getSaldo());
        try {
            Thread.sleep(1000);
        } catch (InterruptedException e) {

            e.printStackTrace();
        }
    }

 }

 @Override
 public void run() {

            this.gestionConcurrente();
 }

}

public class App {

 public static void main(String[] args) {

    GestorCuenta gestor = new GestorCuenta();

    Thread Juan = new Thread(gestor, "Juan");
    Thread Lola = new Thread(gestor, "Lola");

    Juan.start();
    Lola.start();
 }
}
```

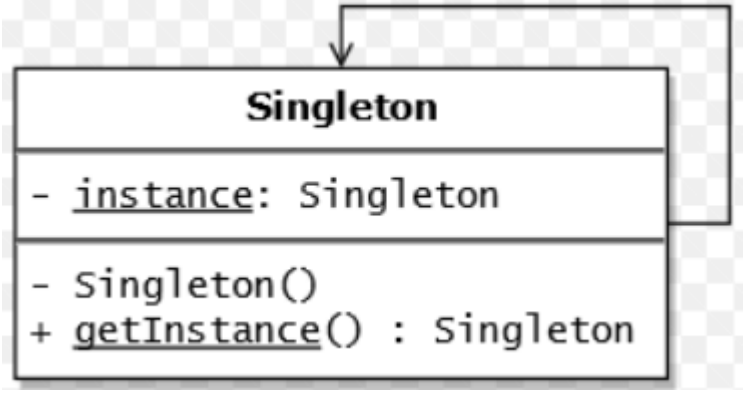
PATRÓN SINGLETON

Cuando estudiamos el patrón *simply factory* indicamos que se trataba de un patrón de diseño creacional y también indicamos que los patrones creacionales se dedican a indicar soluciones a los problemas que surgen cuando se crean objetos. Singleton al igual que Simply factory es un patrón creacional. Pero cada uno de ellos se ocupan de problemas diferentes que ocurren al crear objetos.

Simply factory se ocupaba de decidir que subobjeto se creaba en una jerarquía en la que la superclase solía ser una clase abstracta. Se podían crear en la App muchos objetos.

Por el contrario, Singleton es para garantizar que tan solo exista un objeto de un tipo. Ese objeto va a ser utilizado por diversas partes del código y hay que conseguir que si ya existe un objeto no se cree otro por error.

Una clase que implemente el patrón singleton tendrá la siguiente estructura



Es decir:

- tiene un método público y static getInstance() que devuelve un objeto de la clase. Veremos que getInstance() sólo crea un objeto la primera vez que se invoca luego, devuelve una referencia al objeto previamente creado
  ¡no trabaja igual que el getInstance() de simple factory!
- un constructor privado
- un atributo privado que tiene la referencia al objeto creado
- la clase será *final* para evitar sobreescrituras

Como tiene un método static que se llama típicamente getInstance() recuerda mucho a *simple factory* pero su finalidad es diferente, no tiene nada que ver, aborda otro problema.

En Java el aspecto del uml anterior sería el siguiente

```
class ClaseSingleton {
 private static ClaseSingleton instanciaUnica;

 private ClaseSingleton() {}

 public static ClaseSingleton getInstance() {
    if (instanciaUnica == null) {
       instanciaUnica = new ClaseSingleton();
    }

    return instanciaUnica;
 }
}

class App{
 public static void main(String[] args) {
   //comprobamos que sólo se crea el objeto una vez
   for (int i = 0; i < 5; i++) {
     //ClaseSingleton cs= new ClaseSingleton();  no se puede
     ClaseSingleton cs =ClaseSingleton.getInstance();
     System.out.println(cs);//siempre se va a imprimir la misma referencia
   }
 }
}
```

CONCURRENCIA Y PATRÓN SINGLETON

Además, si resulta que la clase se va a utilizar desde varios hilos, hay que evitar la posibilidad de que por concurrencia se lleguen a crear varias copias. Podría ocurrir dada la ejecución concurrente que dos o más hilos vean que la instancia vale null y se permitiran crear varias instancias. Para evitar esto sincronizamos getInstance()

```
public class ClaseSingleton {
    private static  ClaseSingleton instanciaUnica;
```

```
    private ClaseSingleton() {}

    public static synchronized  ClaseSingleton getInstance() {
        if (instanciaUnica == null) {
            instanciaUnica = new ClaseSingleton();
        }

        return instanciaUnica;
    }
}
```

este tiene soluciones más sofisticadas porque esta solución aunque efectiva puede tener problemas de rendimiento ya que si el código sincronizado consume mucho tiempo de ejecución, provoca largas esperas para otros hilos

Ejemplo sin hilos: observa que aunque por error el código cliente intente crear dos instancias, sólo se crea una

```
final class ConSingleton {
    private static ConSingleton instance;
    public String nombre;

    private ConSingleton(String nombre) {
        this.nombre = nombre;
    }

    public static ConSingleton getInstance(String nombre) {
        if (instance == null) {
            instance = new ConSingleton(nombre);
        }
        return instance;
    }
}
//código cliente que usa la clase Singleton
public class App{
    public static void main(String[] args) {

        ConSingleton uno = ConSingleton.getInstance("UNO");
        ConSingleton dos = ConSingleton.getInstance("DOS");
        System.out.println(uno.nombre);
        System.out.println(dos.nombre);
    }
}
```

salida:
UNO
UNO

Ejemplo con hilos: sin sincronizar el getInstance() todo ocurre tan rápido que se pueden llegar a crear dos instancias.

Si ejecutamos muchas veces el programa(dale muchas veces a play), dependiendo de la gestión de hilos del sistema operativo puede llegue a ocurrir alguna vez que obtengamos de salida
UNO
DOS
o
DOS
UNO
Y también puede ocurrir que un hilo llegue lo suficientemente antes para que sólo se creara una instancia y obtuvieramos

UNO
UNO

o bien
DOS
DOS

```
final class Singleton {
    private static  Singleton instance;
    public String value;

    private Singleton(String value) {
        this.value = value;
    }

    public static Singleton getInstance(String value) {

        if (instance == null) {
            instance = new Singleton(value);
        }
        return instance;
    }
}
//código cliente que usa la clase Singleton

class ThreadUno implements Runnable {
    @Override
    public void run() {
        Singleton singleton = Singleton.getInstance("UNO");
        System.out.println(singleton.value);
    }
}
class ThreadDos implements Runnable {
    @Override
    public void run() {
        Singleton singleton = Singleton.getInstance("DOS");
        System.out.println(singleton.value);
    }
}
public class App{
    public static void main(String[] args) {

        Thread threadUno = new Thread(new ThreadUno());
        Thread threadDos = new Thread(new ThreadDos());
        threadUno.start();
        threadDos.start();
    }
}
```

Ejercicio U5_B9B_E2: soluciona el problema anterior con synchronized