COLECCIONES INMUTABLES Y SINCRONIZADAS

Las colecciones "tradicionales" pueden fallar en entornos concurrentes.Para evitar problemas de concurrencia lo ideal sería utilizar siempre que sea posible colecciones inmutables. Si las colecciones inmutables no nos valen debido a que por ejemplo nuestro programa necesita modificar una colección desde varios hilos, entonces tendríamos que utilizar colecciones sincronizadas.

COLECCIONES INMUTABLES

Aquí reproducimos dos comentarios de Oracle en los que nos explica las bondades de los objetos inmutables (y por tanto de las colecciones) y porque se prefieren a los mutables.

los objetos inmutables son más seguros

https://docs.oracle.com/javase/tutorial/essential/concurrency/immutable.html

Un objeto se considera *inmutable* si su estado no puede cambiar después de su construcción. La máxima confianza en objetos inmutables está ampliamente aceptada como una estrategia sólida para crear código simple y confiable.

Los objetos inmutables son particularmente útiles en aplicaciones concurrentes. Dado que no pueden cambiar de estado, no pueden ser corrompidos por interferencias de subprocesos ni observarse en un estado inconsistente.

Los programadores a menudo son reacios a emplear objetos inmutables, porque se preocupan por el costo de crear un nuevo objeto en lugar de actualizar un objeto en su lugar. El impacto de la creación de objetos a menudo se sobreestima y puede compensarse con algunas de las eficiencias asociadas con los objetos inmutables. Estos incluyen una reducción de la sobrecarga debido a la recolección de basura y la eliminación del código necesario para proteger los objetos mutables de la corrupción.

los objetos inmutables son más eficientes

Como curiosidad, aquí viene un razonamiento al respecto.

https://docs.oracle.com/javase/9/core/creating-immutable-lists-sets-and-maps.htm#JSCOR-GUID-6A9BAE41-A1AD-4AA1-AF1A-A8FC99A14199

CÓMO OBTENER UNA COLECCIÓN INMODIFICABLE/INMUTABLE

A través de métodos static y utilidades

https://docs.oracle.com/javase/9/core/creating-immutable-lists-sets-and-maps.htm#JSCOR-GUID-202D195E-6E18-41F6-90C0-7423B2C9B381

Antes del JDK 9 lo que había era utilidades para convertir una colección en inmodificable con unmodifiableCollection, unmodifiableList, unmodifiableMap y unmodifiableSet. Desde jdk 9 se pueden conseguir realmente inmutables con el método of() que no son más que métodos factoría.

Ejemplo que convierte una lista en inmodificable:

```
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class App {
   public static void main(String[] args) {
      //recuerda que asList genereba una lista de tamaño fijo pero modificable por ejemplo con set()
      List<String> lista = Arrays.asList("a", "b","c");
      lista.set(0,"hola");
      System.out.println(lista);
      lista = Collections.unmodifiableList(lista);
      lista.set(0,"adios"); //ahora GENERA EXCEPCION POR INTENTAR MODIFICAR
   }
}
```

inmodificable puede fallar

En el siguiente ejemplo se observa que con "inmodificable" puedo conservar la referencia a la lista original y es posible intencionadamente o por error modificar la colección con esta referencia. Esto no es posible con of() pues sólo tengo una referencia. En el ejemplo se trabaja con objeto arrayList para hacer un add() pues recuerda que el objeto que devuelve asList() es de tamaño fijo (aunque modificable)

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.List;

public class App {
   public static void main(String[] args) {
      List<String> stringList11 = new ArrayList<>(Arrays.asList("a", "b","c"));
      List<String> stringList12 = Collections.unmodifiableList(stringList11);
      stringList11.add(0,"hola"); //también cambia stringList12
      System.out.println(stringList12);
   }
}
```

Mejor usar métodos of() para obtener auténticos objetos inmutables (Sólo a partir de jdk 9)

Podemos comprobar que para hacer algo parecido a lo anterior sólo disponemos de un método copyof() que realmente genera una copia del original y por lo tanto aunque modifiquemos el original no hay vinculación con el nuevo objeto.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class App {
   public static void main(String[] args) {
        //no se puede cambiar de tamaño ni de valores
        List<Integer> lista = List.of(1,2,3);
        //lista.set(0,99);//salta excepción
        //lista.add(7);//salta excepción
```

```
        List<String> stringList1 = new ArrayList<>(Arrays.asList("a", "b","c"));
        List<String> stringList2 = List.copyOf(stringList1);
        stringList1.add(0,"hola"); //NO cambia stringList12
        System.out.println(stringList2);

    }
}
```

COLECCIONES SINCRONIZADAS

Dos posibilidades:

- convertir una colección tradicional no sincronizada en sincronizada con utilidades como synchronizedCollection, synchronizedList, synchronizedMap y synchronizedSet. Este enfoque es útil si ya hay código que usa colecciones tradicionales y queremos añadir código que las use pero de forma sincronizada
- usar directamente una colección ya sincronizada, esto es importante si la concurrencia es un factor clave ya que las colecciones específicas para la concurrencia son más eficientes, por ejemplo ConcurrentHashMap divide el mapa en varios segmentos y solo bloquea los segmentos relevantes lo que permite a múltiples *threads* acceder a otros segmentos del mismo mapa sin contención. CopyOnWriteArrayList permite varios lectores sin necesidad de sincronización y cuando ocurre una escritura copia el *ArrayList* a uno nuevo. Esta especificidad puede dar más eficiencia ya que no se bloquea toda la colección cada vez que se accede a ella

Ejemplo: El hilo main y un hilo hijo de main comparten un arrayList.

Observa como salta una ConcurrentModificationException

```
import java.util.*;

class App extends Thread {
    static ArrayList<String> l = new ArrayList<>();

    public void run() {
        //esperamos 2 segundos para añadir un elemento
        try {
            Thread.sleep(2000);
        } catch (InterruptedException e) {

        }
        l.add("D");
    }

    public static void main(String[] args) throws InterruptedException {
        l.add("A");
        l.add("B");
        l.add("c");
```

```
        App hiloHijo = new App();
        hiloHijo.start();

       //ahora con un sleep(6000) en cada iteración
       //el hilo main y el hijo coinciden al usar el arrayList
        Iterator<String> itr = l.iterator();
        while (itr.hasNext()) {
            String s = itr.next();
            System.out.println(s);
            Thread.sleep(6000);
        }
        System.out.println(l);
    }
}
```

SOLUCIÓN:
Simplemente sustituyendo la línea de creación del arrayList
`static ArrayList<String> l = new ArrayList<>();`
por
`static CopyOnWriteArrayList<String> l =  new CopyOnWriteArrayList<>();`

Ejecuta de nuevo el ejemplo y observa como ahora no hay error de concurrencia y como el iterador imprime la lista previa modificación

Ocurre lo siguiente:

- El hilo main itera sobre la lista
- tras dos segundos el hilo hijo intenta añadir la lista
- Se crea una copia de esa lista
- El iterador sigue iterando sobre la lista original
- Las modificaciones se hacen sobre la copia
- Al acabar de iterar con la copia modificada actualizamos la referencia a la lista.

Lógicamente, en ciertos contextos puede que no nos encaje esta forma de trabajar de esta clase y entonces habría que usar otra clase o bien programar nosotros personalmente la sincronización a nuestro gusto.

CONCLUSIÓN: Si usamos hilos que comparten colecciones, hay que estudiar el funcionamiento de las clases específicas para entornos concurrentes.