

El ReentrantLock me permite volver a pedir el mismo lock sin tener que bloquearme en el medio. También tiene una opción para garantizar que los que piden el lock esperen en orden. El primero que lo pide es el primero que lo tiene.

El ReadWriteLock está hecho para sistemas con muchas lecturas y pocas escrituras. Se recomienda leer la documentación para saber mejor en qué casos usarlo.

El StampedLock me permite tener versionado sobre un dato, si yo tengo la versión 5 de un dato y la última versión es la 18, tengo que leer de memoria.

# Programación Concurrente - III





# **Alternativas a la sincronización**




# Alternativas a la sincronización

Al tratar de resolver los problemas indicados con anterioridad con sincronización y/o algunas de las otras estrategias vistas se pueden ocasionar otro tipos de **problemas** como **sobre-sincronizar, bloquear threads y/o problemas de memoria.**

Existen estrategias alternativas que en algunos casos pueden ser preferibles ya que nos pueden evitar estos problemas.

Las estrategias que vamos a ver son:


- » Objetos inmutables.
  - » Coordinar mediante una comunicación vía pub/sub.
  - » Hacer sub-procesamientos independientes.
- 



# Objetos inmutables

Un **objeto inmutable** es aquel cuyo **estado no cambia una vez que fue creado.**


Este tipo de objetos **son seguros para aplicaciones concurrentes** ya que **no pueden ser corrompidos mediante la interferencia de Threads,** ni tener problemas de consistencia.





# Objetos inmutables

Para que un **objeto sea inmutable** se puede:

- » Asegurarse que todos los **campos sean privados y finales** (obviamente no proveer **setters**).
  - » Marcar los **métodos con final** (o la clase) para evitar que subclases modifiquen las implementaciones de los métodos.
  - » Asegurarse de tener **referencias a objetos mutables** donde **estos objetos no se modifiquen** en métodos de la clase ni se "escapen" para ser modificados fuera de la clase.
- 

# Ejercicio 1 - Objetos inmutables

1. Bajar el archivo concurrency-iii.zip
2. Correr el SubscriberTest y analizar el resultado
3. Realizar los cambios necesarios a las clases para que Subscriber sea inmutable.

En el ejercicio tenemos un intento de hacer un Objeto inmutable, con objetos final que tampoco proveen setters. Sin embargo, hay un metodo malicioso que se abusa de nuestros getters (que devuelven un puntero a un objeto del estado interno de la clase), y logran cambiarlo.

Por ejemplo, yo puedo poner una lista como final en mi clase, pero cuando devuelvo esa lista en el getter el que lo recibe puede agregar/borrar elementos.

Conclusion: cuando yo quiero hacer que mi clase sea inmutable, tengo que asegurarme que mi contenido tambien sea inmutable. Para solucionar el problema de la lista, una solucion facil es devolver siempre una copia de la misma, aunque esto es ineficiente en memoria.

El metodo List.of me permite devolver una lista inmutable.

Otro problema que surge es cuando recibo una lista por constructor y me guardo esa lista. Cuando la guarde, tendria que guardar una copia de la misma y no la referencia.

# Objetos inmutables


```
public class Subscriber {  
    private final String fullName;  
    private final Date dateOfBirth;  
    public Subscriber(String fullName, Date dateOfBirth) {  
        this.fullName = fullName;  
        this.dateOfBirth = dateOfBirth;  
    }  
  
    public String getFullName() {  
        return fullName;  
    }  
  
    public Date getDateOfBirth() {  
        return new Date(dateOfBirth.getTime());  
    }  
}
```



# Objetos inmutables

Por supuesto, **no todo problema se puede resolver con objetos inmutables.**

Pero es buena práctica empezar haciendo los objetos inmutables, y volverlos mutable a medida que sea necesario.

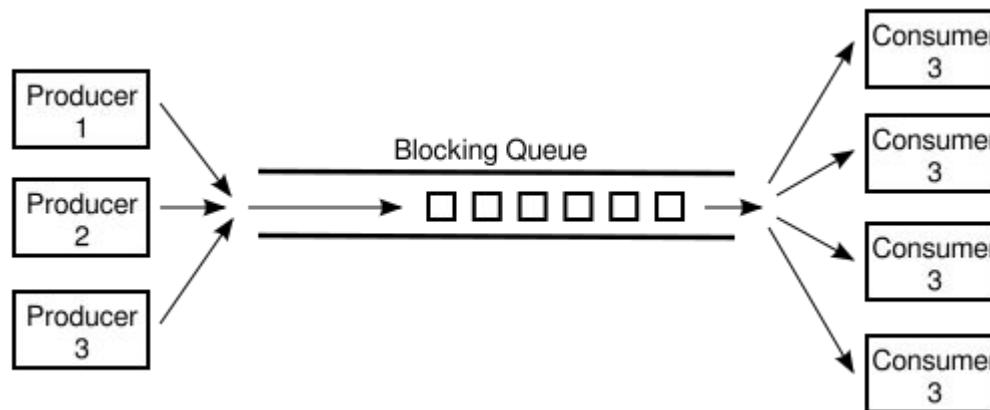




# Pub-Sub

Una manera de coordinar entre Threads que uno dependa del trabajo de otro es armar una estrategia **publisher/subscriber**. La estrategia se basa en


- » Ambos Threads comparten una cola (por eso java.concurrent tiene tantas opciones).
- » Cuando el publisher termina su tarea envía un mensaje mediante la cola.
- » El subscriber lee dicho mensaje de la cola e inicia su tarea.





# Pub-Sub


Esta estrategia permite:

- Reducir la dependencia entre Threads.
  - De acuerdo a la cola utilizada el tiempo de IDLE del Thread se reduce al mínimo.
  - Se reduce el riesgo de errores al no usar notifyAll, o locks bloqueantes.
- 



# Pub-Sub - Queues

De acuerdo a las necesidades el pub sub (y otras estrategias) pueden elegir entre las siguientes colas:

- **ArrayBlockingQueue:** Se implementa sobre un array con tamaño fijo.
  - **LinkedBlockingQueue:** Se implementa sobre un LinkedList sin límite.
  - **PriorityBlockingQueue:** No aplica FIFO sino que ordena por prioridad
  - **DelayQueue:** Los elementos de la lista salen de la misma luego de un delay que cada uno responde
  - **LinkedTransferQueue:** El productor se bloquea al producir hasta que el consumidor toma el elemento.
  - **SynchronousQueue:** Cola que solo acepta 1 elemento.
  - **ConcurrentLinkedQueue:** Cola thread safe con métodos no bloqueantes.
- 



# Pub-Sub - Ejemplos

A la hora de utilizar una estrategia pub/sub se requiere primero definir la cola para comunicarse

```
SynchronousQueue<Integer> queue = new SynchronousQueue<>();
```



# Pub-Sub - Ejemplos

El productor realiza su tarea y se encarga de **publicar en la cola** el mensaje para el/los consumidores.

```
Runnable producer = () -> {  
    Integer producedElement =  
        ThreadLocalRandom  
            .current()  
            .nextInt();  
  
    try {  
        queue.put(producedElement);  
    } catch (InterruptedException ex) {  
        ex.printStackTrace();  
    }  
};
```

# Pub-Sub - Ejemplos

El consumidor lee de la cola y ejecuta en consecuencia.


```
Runnable consumer = () -> {  
    try {  
        Integer consumedElement =  
queue.take();  
        ...  
    } catch (InterruptedException ex) {  
        ex.printStackTrace();  
    }  
};
```



# Pub-Sub - Ejemplos

Ambos elementos son ejecutados usando Threads


```
ExecutorService executor = Executors.newFixedThreadPool(2);  
  
executor.execute(producer);  
executor.execute(consumer);
```





# Pub-Sub

Esta estrategia se porta a sistemas distribuidos donde es común utilizar pub-sub como estrategia de comunicación entre componentes remotos.





## Ejercicio 2 - Pub Sub

1. Analizar la ejecución de PubSubTest.
2. Modificar el consumer para que imprima la suma de los números recibidos al terminar su ejecución.


En este ejemplo se muestra el uso del PoisonPill. La idea es que cuando el Producer pone un poison pill en la lista, entonces el Consumer sabe que ya no hay nada mas que consumir.

PoisonPillPerProducer y el modulo es porque el numero de Consumers puede no ser el mismo que el numero de Producers, estos dos se encargan de que se produzcan el numero exacto de PoisonPills para matar a todos los Consumers.



## Subdivisión de la Tarea

Otra estrategia de realizar procesamiento de grandes datos de información es (de ser posible):

- **Particionar la información** entre unidades de procesamiento (workers o threads).
  - Cada worker **calcule un resultado parcial** sobre su partición
  - Luego **con los resultados** de todas las particiones **se calcula el resultado total.**
- 



## Subdivisión de la Tarea

Esta estrategia también es deseable porque:

- **No hay riesgo de interferencia** (cada partición es independiente del resto y debería ser inmutable).
- Se puede **optimizar el procesamiento** porque se procesa un subset de la data
- Se puede **reutilizar workers** si las particiones son más que los workers.

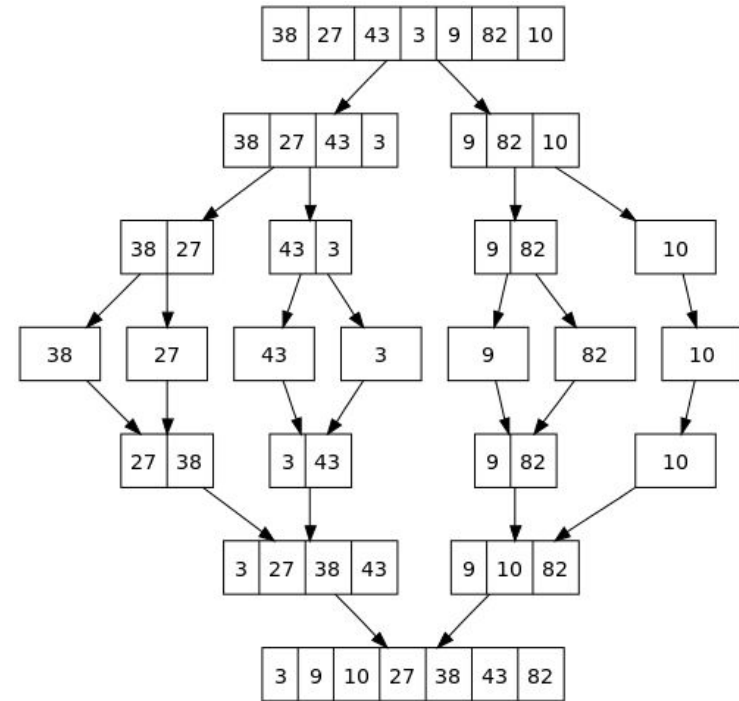
La principal contra es lograr describir la tarea para particionar la información.



# Subdivisión de la Tarea

Como **ejemplo** de esta tarea podemos ver un algoritmo de ordenamiento como **Mergesort**.

Cuyo algoritmo requiere particionar la data e ir ordenando dentro de las particiones. La tarea dentro de cada partición es independiente de las otras particiones.




6 5 3 1 8 7 2 4



## Subdivisión de la Tarea

Esta es otra estrategia que se porta a sistemas distribuidos ya que el particionamiento de datos es deseable cuando tengo workers en nodos diferentes.



# Subdivisión de la Tarea

En **Java 8** se incluye nativamente la **posibilidad de paralelizar los Streams**.

Esto hace que el cálculo de las operaciones se realicen en Threads obtenidos mediante al ForkJoinPool.

La división y generación de Threads la realiza la JVM. Esto trae sus riesgos porque **al ser una solución genérica no siempre es la más óptima**.

```
double average = roster
    .parallelStream()
    .filter(p -> p.getGender() == Person.Sex.MALE)
    .mapToInt(Person::getAge)
    .average()
    .getAsDouble();
```

## Ejercicio 3 - Parallel sort

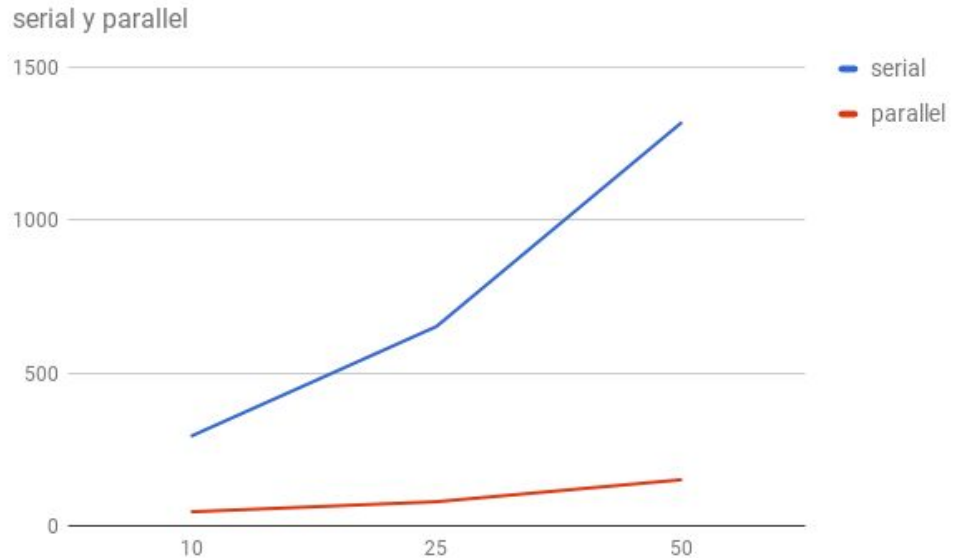
La idea es comparar una ejecución de ordenamiento paralela y otra secuencial. Para ello vamos a un test que:

- » Genere 3 arrays de enteros con números aleatorios con diferentes tamaños (10 M, 25M y 50M de números)
- » Por cada array saque el promedio de tiempo de 4 ejecuciones de ordenamiento para:
  - Arrays#parallelSort
  - Arrays#sort
- » Imprima a salida estándar el resultado de cada una de las ejecuciones

## Ejercicio 3 Parallel sort

En una corrida el resultado dio:

- » 10M serial: 293 ms
- » 10M parallel: 46 ms
- » 25M serial: 652 ms
- » 25M parallel: 79 ms
- » 50M serial: 1320 ms
- » 50M parallel: 151 ms



Las ejecuciones concurrentes son mejores a medida que el número de valores crece.

Existe la posibilidad de que para valores muy chicos el serial sea igual o hasta mejor que paralelo (probar con 10K números) ¿Por qué?

Es verdad. Si estoy manejando pocos valores (en el orden de cientos o pocos miles), no tiene sentido paralelizar, ya que estaría perdiendo mas tiempo. Este error se llama sobrediseñar.





## Parallel Stream - Orden

Al ejecutar una acción en paralelo la JVM decide cómo dividir la secuencia en partes buscando la manera más eficiente de dividir para la ejecución paralela.

Internamente utiliza la clase *Split Iterator*.

Al no estar garantizado el orden, además de los problemas de concurrencia, los lambdas que se pasan como operaciones no deben:

- Tener side-effects: modificar valores de la clase que los contiene.
  - Ser stateful: guardar estado interno.
- 



# Scheduled Future






# Scheduled Futures

Los scheduled futures son futures que se corren de manera periódica o con algún delay (o ambos).

La interfaz es la misma que Future más la incorporación de Delayed

```
public interface ScheduledFuture<V> extends Delayed, Future<V> {  
}
```

```
public interface Delayed extends Comparable<Delayed> {  
    long getDelay(TimeUnit var1);  
}
```





# Scheduled Futures

Para instanciar un ScheduledFuture existe un executor service

```
final ScheduledExecutorService executorService = Executors  
    .newSingleThreadScheduledExecutor();
```



# Scheduled Futures

Y a partir del mismo se puede instanciar una tarea que se ejecute luego de un cierto delay

```
Future<String> resultFuture =  
    executorService.schedule(callableTask, 1, TimeUnit.SECONDS);
```

Esta tarea se corre 1 vez luego de 1 segundo de delay



# Scheduled Futures

Para instanciar una tarea que se repite existen dos métodos

```
executorService.scheduleAtFixedRate(runnableTask, 100, 450,  
TimeUnit.MILLISECONDS);
```

Luego de 100 Ms corre una tarea cada 450 Ms a menos que la tarea tarde más

```
executorService.scheduleWithFixedDelay(task, 100, 150,  
TimeUnit.MILLISECONDS);
```

Luego de 100 Ms corre una tarea y se repite 150 Ms después de que la tarea termine.

En ambos casos las tareas terminan al finalizar el ExecutorService o se la cancele

## Ejercicio 4 - Scheduled

1. Implementar ScheduledTest
2. El mismo debe correr una tarea que corra cada 10 segundos e imprima el timestamp del momento.
3. Además debe ejecutar una segunda tarea con un delay de 60 segundos que cancele la primera tarea



# Completable Future



# Completable Futures

Representan un Future que puede ser completado programáticamente y puede ser encadenado y combinado con otros CompletableFutures debido a que implementa la interfaz **CompletionStage**.

Construcción:

- A pesar de que existe un constructor default para poder crear un objeto y luego completarlo:

```
CompletableFuture<String> cf = new CompletableFuture<>();
```

```
//completar el future  
cf.complete("resultado correcto");
```

```
//o si quiero tirar error  
cf.completeExceptionally(new IllegalArgumentException("este  
future no termino bien"));
```

# Completable Futures

- Existen métodos estáticos de construcción de acuerdo al tipo de acción que se quiere pasarle.

```
//tarea ya terminada y con respuesta
CompletableFuture.completedFuture("resultado");

//tarea como runnable
CompletableFuture.runAsync(() -> System.out.println("tarea"));

//tarea como un supplier (similar a Callable)
CompletableFuture.supplyAsync(() -> "resultado");
```

run y supply utilizan el `forkJoinPool` a menos que se use las versiones que reciben un `ExecutorService`.

# Completable Futures - Chaining

Se puede obtener un `CompletableFuture` que es el encadenamiento entre un `CompletableFuture` y una acción que corre después de que finalizó el `CompletableFuture`.

Donde la acción puede ser:

Un `Runnable` que corre sin importar la respuesta del Future inicial.

```
CompletableFuture<Void> thenRun = cf.thenRun(() ->
System.out.println("tarea"));
```

Un `Consumer` que utiliza la respuesta del mismo.

```
CompletableFuture<Void> thenAccept = cf.thenAccept(response ->
System.out.println(response));
```

# Completable Futures - Chaining

Una función que transforma la respuesta del primero en otro valor.

```
CompletableFuture<String> thenApply = cf.thenApply(response  
-> response.toLowerCase());
```

Una función que transforma la respuesta del primero en un CompletableFuture.

```
CompletableFuture<String> thenCompose =  
cf.thenCompose(response ->  
    CompletableFuture.completedFuture(response.toLo  
werCase()));
```

Estas versiones corren en el mismo thread pero existen variantes **\*Async** para correr en threads diferentes usando ForkJoinPool o un ExecutorService

# Completable Futures - On Error

Al encadenar puede que alguna de las acciones termine con un error, es decir, arroja una excepción. Para estos casos se puede encadenar un manejo de dicho caso.

Se puede manejar solamente la excepción mediante una función que transforme la excepción en un valor.

```
CompletableFuture<String> exceptionally =  
    cf.exceptionally(th -> th.getMessage());
```

# Completable Futures - On Error

O una función que consume tanto el caso ok como el caso de error (pero lo deja para futuros encadenamientos).

```
CompletableFuture<String> whenComplete = cf.whenComplete((r, e) ->
{
    if (r != null) {
        System.out.println(r);
    } else if (e != null) {
        System.out.println(e.getMessage());
    } else {
        System.out.println("don't know what happened");
    }
});
```

Sólo uno tiene  
valor != `

Solo uno tiene valor != NULL

# Completable Futures - Combine

Se puede encadenar un `CompletableFuture` a la ejecución de más de un `CompletableFuture`

- Se puede **combinar dos ejecuciones** para aplicar una función al resultado de las dos:

```
ExecutorService service = Executors.newCachedThreadPool();

CompletableFuture<String> nameTask =
    CompletableFuture.supplyAsync(() -> "name", service);
CompletableFuture<String> subscribersTask =
    CompletableFuture.supplyAsync(() -> "2000", service);

CompletableFuture<String> combined =
    nameTask.thenCombineAsync(subscribersTask, (f, s) -> f + " with: " + s,
    service);
```

# Completable Futures - Combine


O aplicar una función al resultado de la primera ejecución que termine:

```
CompletableFuture<String> nameTask =  
    CompletableFuture.supplyAsync(() -> "name");  
  
CompletableFuture<String> username =  
    CompletableFuture.supplyAsync(() -> "username");  
  
CompletableFuture<String> either =  
    nameTask.applyToEither(username, (r) -> "obtained: " + r);
```






# Para finalizar

- Además de las estructuras vistas en la clase anterior existen otras estrategias para solventar problemas de Thread Safety como el manejo de **objetos inmutables, pub/sub y subdivisión de tareas**.
  - Varias de estas estrategias las volveremos a ver al pasar a programación distribuida.
  - También se pueden utilizar ejecuciones más específicas cuando las tareas tengan características especiales como la necesidad de subdividir la tarea, o correrlas periódicamente
- 



# Referencias y para profundizar

- » <https://docs.oracle.com/javase/tutorial/essential/concurrency/index.html>
  - » **Concurrent Programming in Java: Design Principles and Pattern (2nd Edition)** by Doug Lea. A comprehensive work by a leading expert, who's also the architect of the Java platform's concurrency framework.
  - » **Java Concurrency in Practice** by Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea. A practical guide designed to be accessible to the novice.
  - » **Effective Java Programming Language Guide (2nd Edition)** by Joshua Bloch. Though this is a general programming guide, its chapter on threads contains essential "best practices" for concurrent programming.
- 



# CREDITS

Content of the slides:

- » POD - ITBA

Images:

- » POD - ITBA
- » Or obtained from: [commons.wikimedia.org](https://commons.wikimedia.org)

Slides theme credit:

Special thanks to all the people who made and released these awesome resources for free:

- » Presentation template by [SlidesCarnival](https://slidescarnival.com)
  - » Photographs by [Unsplash](https://unsplash.com)
- 