



STM – Memoria Transaccional Software

CONTENIDO:

- Deficiencias del Modelo de Objetos
- Separación Identidad-Estado
- Por Qué Utilizar STM
- Concepto de Transacción. Memoria Transaccional
- El Lenguaje Clojure
- Transacciones con Clojure
- Ejemplos
- STM en Java sobre Clojure

REFERENCIAS:

Subramanian, V. Programming Concurrency on the JVM. The Pragmatic Bookshelf, 2001.

Concurrencia sin Dolor en Java Puro (parte 2):

http://www.javamexico.org/blogs/ezamudio/concurrencia_sin_dolor_en_java_puro_parte_2

Clojure Home Page: <http://clojure.org/>



ESCENARIO ACTUAL

El presente y el futuro giran en torno a procesadores multinúcleo:

- Mayor número de instrucciones por segundo
- Rendimiento superior en problemas paralelizables.

Pero la programación paralela es más compleja:

- Algoritmos complejos
- Depuración
- No determinismo
- Escalabilidad
- Rendimiento no asegurado
- Memoria compartida



Deficiencias del Modelo de Objetos

- Los lenguajes OO fusionan tercamente identidad de un objeto (quién) con su contenido (qué)
- Acceder a un objeto vía referencia es acceder al estado
- Mutar estados mediante referencias se considera "normal"
- El estado puede cambiar en cualquier momento
- Eso hace que la identidad sea mutable (no debería serlo)
- Múltiples hilos entran a la identidad por la misma referencia
- Necesitamos bloquear los hilos
- El grado de concurrencia/paralelismo se reduce



STM – Memoria Transaccional Software

MOTIVACIÓN

Obtener nuevas estructuras de datos:

- Con operaciones que no requieran exclusión mutua (libres de bloqueos).
- Que eviten, por tanto, los problemas derivados de bloqueos:
 - Inversiones de prioridad.
 - Interbloqueos.
 - Condiciones de concurso continuas sobre el mismo cerrojo.

Simplificar la programación paralela ejecutando atómicamente un conjunto de instrucciones (transacción).



Memoria Transaccional

- Mecanismo de control de la concurrencia análogo al sistema de transacciones de las bases de datos para controlar el acceso a la memoria compartida.
- Características
 - Código sencillo.
 - Mezcla operaciones de grano fino y grueso.
 - Más difícil de implementar que los cerrojos.



Concepto de Transacción

- Una transacción es una secuencia finita de instrucciones englobadas en un bloque cuya operación es completa.
- Propiedades **ACID**:
 - *Atomicity* (atomicidad)
 - *Consistency* (consistencia)
 - *Isolation* (aislamiento)
 - *Durability* (durabilidad)

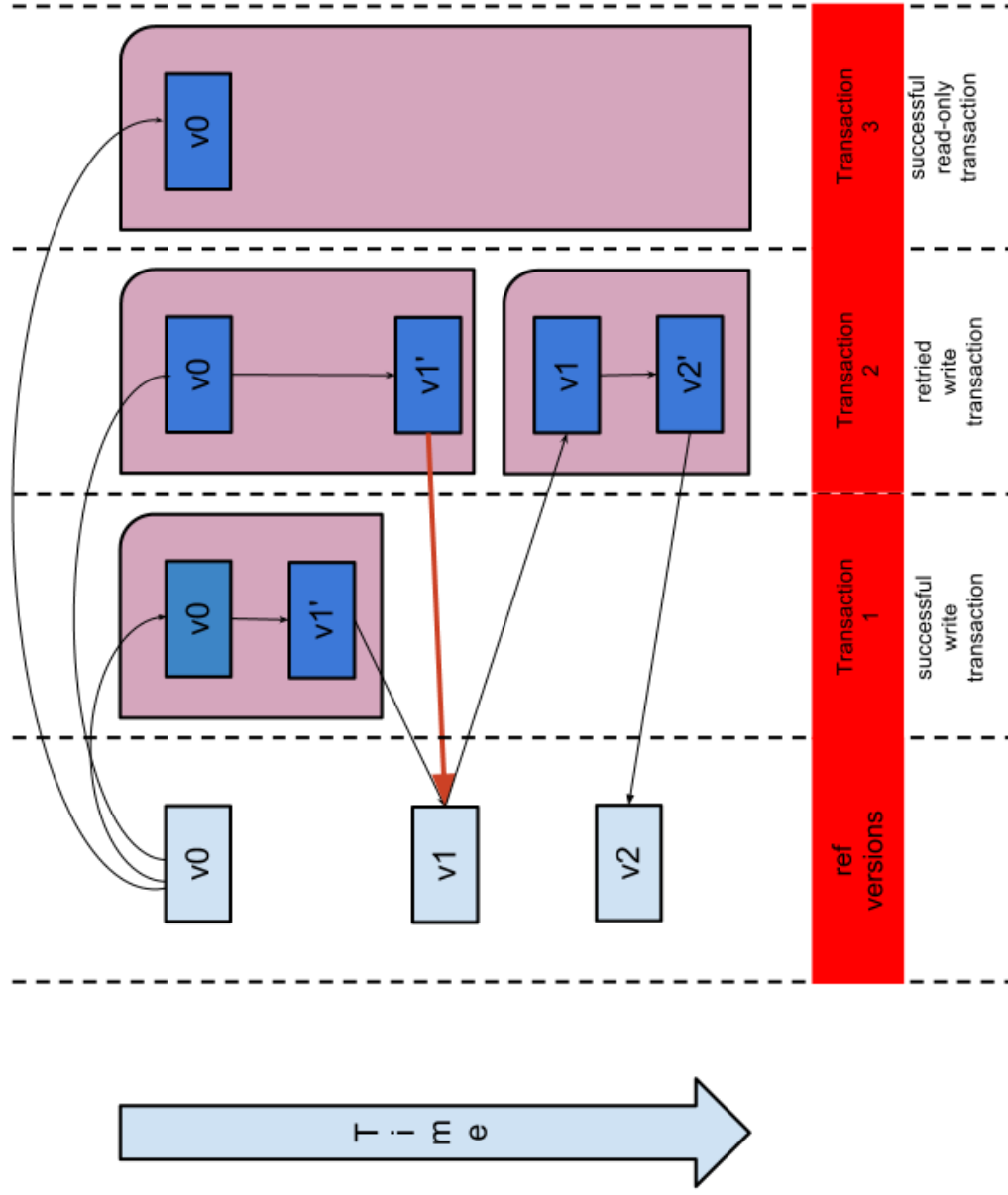
La consistencia, aislamiento y durabilidad pueden aparecer fusionadas bajo el concepto de serialización.



Funcionamiento de una Transacción

Algoritmo de procesamiento de una Transacción

1. Inicio
2. Hacer copia privada de los datos compartidos
3. Hacer actualizaciones en la copia privada
4. Ahora:
 - 4.1 Si datos compartidos no modificados->actualizar datos compartidos con copia privada y goto(Fin)
 - 4.2 Si hay conflictos, descartar copia privada y goto(Inicio)
5. Fin





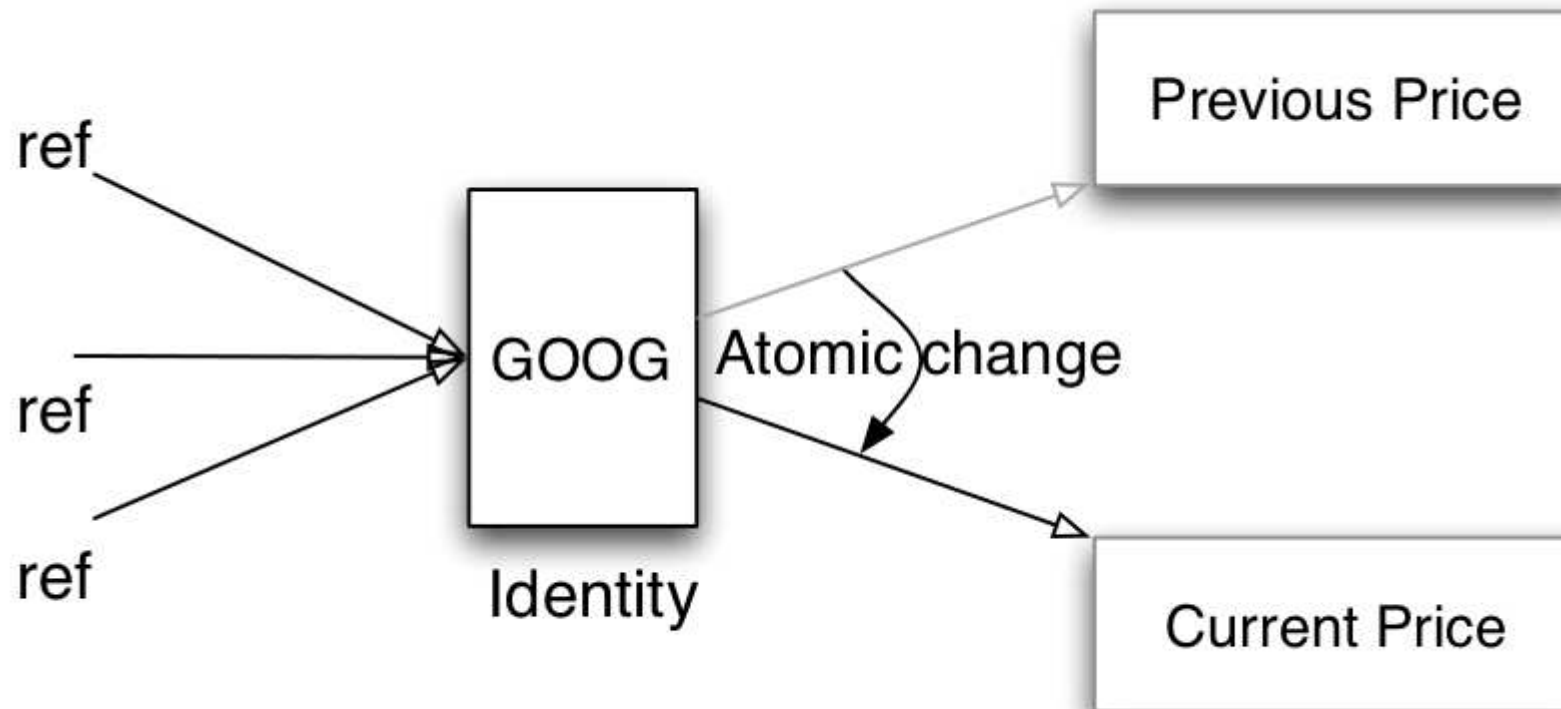
STM Implementaciones

- Existen implementaciones en Hardware, Software e Híbridas.
- Nosotros veremos las implementaciones en Software:
 - Mayor variedad de algoritmos y más sofisticados.
 - Fácilmente modificable y evolucionable.
 - Alta capacidad de integración con sistemas existentes.

Separación de Estado e Identidad

- La orientación a objetos no modela el mundo real.
- En la realidad, el objeto como identidad está separado del estado instantáneo de los valores que lo componen.
- Ejemplo: el precio en bolsa de las acciones de Google el 10/12/2010 era 592,21\$. Este valor es **inmutable** independientemente del valor actual de las acciones.

Separación de Estado e Identidad





Clojure

- Clojure es un lenguaje **funcional**, diseñado para ejecutarse sobre la JVM.
- No hay punto y coma, ni llaves, ni siquiera es necesario usar la coma. Se utilizan los paréntesis (casi) de la misma forma que en la notación funcional.
- Los operadores se colocan al principio, al igual que las funciones.
- La sintaxis de Clojure se resume en
(**function** param1 param2 ... paramN)

Clojure

- Un programa Clojure está compuesto de listas, donde el primer elemento es evaluado y el resto son tomados como argumentos.
- Prueba:
 - (+ 4 6)
 - (/ 5 3)



Clojure

- Muchas funciones en Clojure aceptan un número indeterminado de parámetros.
- Prueba:
 - `(+ 4 6 1 2 2 3 5 6 4)`

Clojure

- Para definir funciones, usamos **defn**, seguido del nombre de la función, los argumentos y el cuerpo de la función.
- Prueba:
 - `(defn suma [x y] (+ x y))`
 - `(suma 4 6)`

Clojure

- En Clojure, las funciones son un tipo más. Pueden ser definidas de forma anónima con `fn`, pueden ser devueltas por otra función y evaluadas de inmediato.
- Prueba:
 - `((fn [x y] (- x y)) 5 2)`

Closure

- **defn** es un “sinónimo” para **def** y **fn**. **fn** declara la función, y **def** crea una referencia en forma de nombre para ella.
- Cuando definimos la suma, podríamos haberlo hecho así:
 - `(def suma (fn [x y] (+ x y)))`
 - `(suma 4 6)`

Clojure - Colecciones

- Ya conocemos las listas en Clojure, y sabemos que, por defecto, siempre se evalúa el primer elemento.
- Para mantener la lista como tal, hay que escaparla. Se usa el apóstrofo (').
 - Lista: `'(1 2 3 4 5 6 7)`
 - Vector: `[1 2 3 4 5 6]`
 - Diccionario (map): `{:nombre "Pepe" :age 25}`
 - Conjunto (set): `#{1 2 3 4 5 6}`

Clojure - Ejemplos

Clojure	Equivalente Java
<code>(not k)</code>	<code>!k</code>
<code>(inc a)</code>	<code>a+1</code>
<code>(/ (+ x y) 2)</code>	<code>(x + y) / 2</code>
<code>(instance? java.util.List a)</code>	<code>a instanceof java.util.List</code>
<code>(if (not a) (inc b) (dec b))</code>	<code>!a ? b+1 : b-1</code>
<code>(Math/pow 2 10)</code>	<code>Math.pow(2, 10)</code>
<code>(.meth Obj "asd" (.meth2 Obj2))</code>	<code>Obj.meth("asd", Obj2.meth2())</code>



Clojure

- `def` no solo vale para funciones, sino que se puede usar con cualquier tipo. Nosotros lo usaremos para trabajar con `ref`.
- Todos los valores en Clojure son inmutables, y las identidades solo pueden cambiar en transacciones.
- `ref` se encarga de crear identidades mutables.

Clojure: Transacciones STM

- Una transacción se crea envolviendo el código en un bloque **dosync**, de la misma forma que hacíamos con **synchronized** en java. OJO: ¡NO ES LO MISMO!
- Una vez dentro del bloque, podemos cambiar el estado de una identidad mediante **ref-set**, **alter** o **commute**.



Clojure: Modificación de Identidades

- **ref-set**: establece el valor de la identidad y lo devuelve.
- **alter**: establece el valor de la identidad como el resultado de aplicar una función y lo devuelve. Suele ser la más usada.
- **commute**: permite *relajar* las transacciones. Aplica una función al último valor que ha tenido la identidad, en vez de al valor que tenía al comenzar la transacción. Es útil cuando podemos obviar el orden en el que se realizan los cambios, y provee mayor concurrencia que **alter**.



Ejemplo 1: mutate.clj

```
(def balance (ref 0))  
(println "El saldo es " @balance)  
(ref-set balance 100)  
(println "El saldo es ahora " @balance)
```

Ejemplo 1: mutatesuccess.clj

```
(def balance (ref 0))  
(println "El saldo es " @balance)  
(dosync  
  (ref-set balance 100))  
(println "El saldo es ahora " @balance)
```





Clojure: Concurrencia

- Las transacciones tienen que ser idempotentes(*): no sabemos el número de veces que se van a ejecutar.
- En el ejemplo anterior, cambiamos el saldo de una cuenta bancaria. Ahora vemos que pasa cuando múltiples transacciones compiten por cambiar una misma identidad.


(*) Sin embargo, en los ejemplos que te presentamos se usan impresiones por pantalla. Es necesario para ilustrar el comportamiento. Recuerda no hacer lo mismo.

Ejemplo 3: concurrentChangeToBalance.clj

```
(defn deposit [balance amount]
  (dosync
    (println "Listo para ingresar... " amount)
    (let [current-balance @balance]
      (println "Simulando retraso...")
      (. Thread sleep 2000)
      (alter balance + amount)
      (println "Realizado el ingreso de " amount))))
```



```
(defn withdraw [balance amount]
  (dosync
    (println "Listo para reintegro... " amount)
    (let [current-balance @balance]
      (println "Simulando retraso...")
      (. Thread sleep 2000)
      (alter balance - amount)
      (println "Realizado reintegro de " amount)))))
```



```
(def balance1 (ref 100))

(println "El saldo es " @balance1)

(future (deposit balance1 20))
(future (withdraw balance1 10))
(. Thread sleep 10000)

(println "El saldo es ahora " @balance1)
```



Clojure: Concurrencia en Colecciones

- ¿Qué ocurre cuando tenemos colecciones, por ejemplo, una lista?
- Las colecciones también son inmutables. Sin embargo, podemos definir, como antes, identidades mutables con las que simular que la lista cambia aunque lo único que hacemos es *cambiar el punto de vista*.

Ejemplo 4: concurrentListChange.clj

```
(defn add-item [wishlist item]
  (dosync (alter wishlist conj item)))

(def family-wishlist (ref ("iPad")))
(def original-wishlist @family-wishlist)

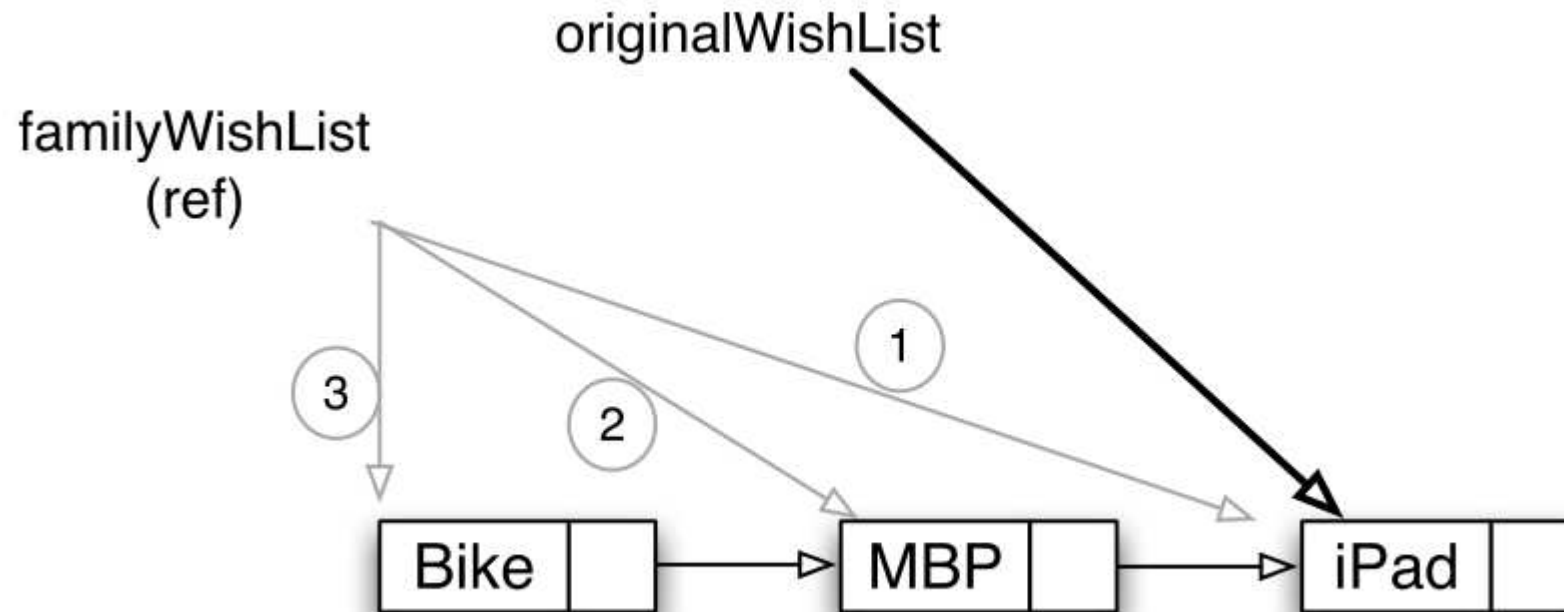
(println "Lista de deseos original: " original-wishlist)

(future (addItem family-wishlist "MBP"))
(future (addItem family-wishlist "Bike"))

(. Thread sleep 1000)

(println "Original wish list is" original-wishlist)
(println "Updated wish list is" @family-wishlist)
```

Separación de Estado e Identidad en Colecciones




Clojure: Referencias de Sólo Lectura

- Ya hemos visto como Clojure gestiona los conflictos entre transacciones que compiten por modificar una misma identidad.
- ¿Qué ocurre cuando hay restricciones cruzadas?
 - Imaginemos que tenemos dos cuentas: la cuenta de ahorro y la cuenta corriente. El banco nos pone una restricción: no podemos tener menos de 1000€ entre las dos cuentas. Ilustremos este ejemplo.

Ejemplo 5: writeSkew.clj

```
(defn withdraw-account [from-balance constraining-balance amount]
  (dosync
    (let [total-balance (+ @from-balance @constraining-balance)]
      (. Thread sleep 1000)
      (if (>= (- total-balance amount) 1000)
        (alter from-balance - amount)
        (println "Sorry, can't withdraw due to constraint
violation")))))
```



```
(def checking-balance (ref 500))
(def savings-balance (ref 600))
(println "checking-balance is" @checking-balance)
(println "savings-balance is" @savings-balance)
(println "Total balance is" (+ @checking-balance @savings-balance))

(future (withdraw-account checking-balance savings-balance 100))
(future (withdraw-account savings-balance checking-balance 100))

(. Thread sleep 2000)
(println "checking-balance is" @checking-balance)
(println "savings-balance is" @savings-balance)
(println "Total balance is" (+ @checking-balance @savings-balance))
```

Clojure: Referencias de Sólo Lectura

- Estos problemas son fáciles de solucionar haciendo uso de ensure.
- Indicamos así a Clojure que *le eche un ojo* a una variable que solo leemos y en ningún caso modificamos. La STM se encargará de reintentar el bloque si este valor es modificado fuera de la transacción, antes de que esta acabe.

Ejemplo 6: writeSkew.clj

```
(defn withdraw-account [from-balance constraining-balance amount]
  (dosync
    (let [total-balance (+ @from-balance
                           (ensure constraining-balance))]
      (. Thread sleep 1000)
      (if (>= (- total-balance amount) 1000)
        (alter from-balance - amount)
        (println "Sorry, can't withdraw due to constraint
violation")))))
```




STM en Java sobre Clojure

- Clojure está diseñado para ejecutarse sobre la máquina virtual de Java.
- Por ello, podemos usar la API de Clojure directamente desde Java.
- En particular, nos interesan la clase `Ref` y `LockingTransaction`
- `LockingTransaction` posee el método `runInTransaction()`, que recibe un objeto `Callable`.


Ejemplo 7: Account.java

```
public class Account
{
    final private Ref saldo;
    public Account(final int saldoInicial) throws Exception
    {
        saldo = new Ref(saldoInicial);
    }

    public int getSaldo()
    {
        return (Integer)saldo.deref();
    }
}
```




```
public void deposito(final int cantidad) throws Exception
{
    LockingTransaction.runInTransaction(new Callable<Boolean>()
    {
        public Boolean call()
        {
            if (cantidad > 0)
            {
                final int saldoActual = (Integer)saldo.deref();
                saldo.set(saldoActual + cantidad);
                System.out.println("Deposito de " + cantidad + " hecho");
                return true;
            }
            else
                throw new RuntimeException("Operacion invalida");
        }
    });
}
```



```
public void reintegro(final int cantidad) throws Exception
{
    LockingTransaction.runInTransaction(new Callable<Boolean>()
    {
        public Boolean call()
        {
            final int saldoActual = (Integer)saldo.deref();
            if (cantidad > 0 && saldoActual >= cantidad)
            {
                saldo.set(saldoActual - cantidad);
                System.out.println("Reintegro de " + cantidad + " hecho");
                return true;
            }
            else
                throw new RuntimeException("Operacion invalida");
        }
    });
}
```



Usando la Cuenta en Modo Seguro: Transfer.java

```
public class Transfer
{
    public static void transfer(final Account cuentaOrigen,
                                final Account cuentaDestino,
                                final int cantidad) throws Exception
    {
        LockingTransaction.runInTransaction(new Callable<Boolean>()
        {
            public Boolean call()
            {
                cuentaDestino.deposito(cantidad);
                cuentaOrigen.reintegro(cantidad);
                return true;
            }
        });
    }
}
```



```
public static void transferirYmostrar(final Account cuentaOrigen,
                                     final Account cuentaDestino,
                                     final int cantidad)
{
    try
    {
        transfer(cuentaOrigen, cuentaDestino, cantidad);
    }
    catch (Exception e)
    {
        System.out.println("Transferencia fallida: " + ex);
    }

    System.out.println("Saldo cuenta origen: " + cuentaOrigen.getSaldo());
    System.out.println("Saldo cuenta destino: " + cuentaDestino.getSaldo());
}
```



```
public static void main(String[] args) throws Exception
{
    final Account cuenta1 = new Account(2000);
    final Account cuenta2 = new Account(100);

    transferirYmostrar(cuenta1, cuenta2, 500);
    transferirYmostrar(cuenta1, cuenta2, 5000);
}
```