Bank Exercise Discussion

1. Storing accounts in a Map
2. Returning account numbers
3. Generating unique account numbers
4. Accessing the balance value
5. Visibility
6. The difficulty of implementing a correct transfer method

Problem 1: Map

- **Map<String,Account> accounts = new HashMap<>();**

```
package bank.local;
class Bank implements bank.Bank {
   private Map<String, Account> accounts = new HashMap<>();
```

  - Problem: HashMap and TreeMap are not thread safe
    - simultaneous (read & write) access may not work correctly, e.g. due to
      - rehashing operations
      - tree rebalancing operations
  - Solution: Synchronization

> **Note that this implementation is not synchronized.** If multiple threads access a hash map concurrently, and at least one of the threads modifies the map structurally, it *must* be synchronized externally. (A structural modification is any operation that adds or deletes one or more mappings; merely changing the value associated with a key that an instance already contains is not a structural modification.)

Problem 1: Map

- **Synchronization of all methods which access accounts**

```
public synchronized String createAccount(String owner)
public synchronized Account getAccount(String number)
public synchronized Set<String> getAccountNumbers()
public synchronized void closeAccount(String number)
```

- **Synchronization of collection access only**

```
synchronized(this) { accounts.add(a); }
```

```
synchronized(accounts) { accounts.add(a); }
```

Problem 1: Map

- **Synchronization wrapper:**

```
accounts = Collections.synchronizedMap(
                                    new HashMap<String,Account>());
```

```
private static class SynchronizedMap<K,V> implements Map<K,V> {
   private final Map<K, V> m; // Backing Map
   final Object mutex; // Object on which to synchronize
   SynchronizedMap(Map<K, V> m) { this.m=m; this.mutex=this; }
   public int size() {
      synchronized(mutex) { return m.size(); }
   }
   public V get(Object key) {
      synchronized(mutex) { return m.get(key); }
   }
   public V put(K key, V value) {
      synchronized (mutex) { return m.put(key, value); }
   }
}
```

Problem 1: Map

- **java.util.concurrent.ConcurrentXXXMap**

```
accounts = new ConcurrentHashMap<>();
accounts = new ConcurrentSkipListMap<>();
```

  - Concurrent read and writes are possible
    - Uses lock striping => JCIP 5.2.1
  - With the constructor a concurrency level is specified (default = 16)
    - The expected number of concurrently updating threads
    - The implementation performs internal sizing to try to accommodate this many threads
  - Implements interface ConcurrentMap

Problem 1: Map

- **java.util.concurrent.ConcurrentMap**
  - Provides additional atomic compound actions

```
public interface ConcurrentMap<K, V> extends Map<K, V> {
    V putIfAbsent(K key, V value);
    boolean remove(Object key, Object value);
    boolean replace(K key, V oldValue, V newValue);
    V replace(K key, V value);
}
```

  - putIfAbsent

```
if (!map.containsKey(key))
  return map.put(key, value);
else
   return map.get(key);
```

Problem 2: getAccountNumbers

- **Fail-Fast Iterators**

```
public Set<String> getAccountNumbers() {
   Set<String> numbers = new HashSet<String>();
   for(String s : accounts.keySet()) {
      if(accounts.get(s).isActive()) numbers.add(s);
   }
   return numbers;
}
```

  - Iteration over *accounts.keySet* may throw a *ConcurrentModification-Exception* if an account is added/removed simultaneously
  - Holds also for *new HashSet<String>(accounts.keySet())*
- **Solution**
  - Synchronize Iteration on the same lock as the modification operations
  - Use a *ConcurrentMap* implementation
    - Their Iterators guarantee to not throw a ConcurrentModificationException

Problem 3: Unique Account Number

- **Account**

```
class Account implements bank.Account {
   private static int id = 0;

   private String number; // unique account number
   private String owner;
   private boolean active;

   Account(String owner){
      this.owner  = owner;
      this.number = "101-47-00" + id++;
   }
   ...
```

  - Problem 1: `id++` is not an atomic operation
  - Problem 2: updates made on `id` may not be visible by all threads

Problem 3: Unique Account Numbers: Solution

- **Account**

```
class Account implements bank.Account {
   private static int id = 0;
   private static synchronized int getNewId() {
      return id++;
   }
   private String number; // unique account number
   private String owner;
   private boolean active;

   Account(String owner){
      this.owner  = owner;
      this.number = "101-47-00" + getNewId();
   }
   ...
```

Problem 3: Unique Account Numbers: Solution

- **Account**

```
class Account implements bank.Account {
   private static final AtomicInteger id =
                                        new AtomicInteger(0);

   private String number; // unique account number
   private String owner;
   private boolean active;

   Account(String owner){
      this.owner  = owner;
      this.number = "101-47-00" + id.getAndIncrement();
   }
   ...
```

Problem 3: Unique Account Numbers: Solution

- **Account**

```
class Account implements bank.Account {
   private String number; // unique account number
   private String owner;
   private boolean active;

   Account(String owner){
      this.owner  = owner;
      this.number = java.util.UUID.randomUUID();
   }
   ...
```

- **Thread-safe since Java 7**
  - http://bugs.sun.com/view_bug.do?bug_id=6611830

Problem 4: Accessing doubles

- **Account.getBalance**

```
class Account implements bank.Account {
   private double balance = 0.0;

   public double getBalance() {
      return balance;
   }
```

  - Problem 1:
    - reading 64bit (double) is not atomic, `getBalance` may return inconsistent values for the balance
  - Problem 2:
    - updated values of balance may not be visible to calling thread, i.e. the client might see an outdated value

Problem 4: Accessing doubles: Solutions

- **Synchronized Access**

```
class Account implements bank.Account {
   private double balance = 0.0;
   public synchronized double getBalance() {
      return balance;
   }
```

- **Volatile Access**

```
class Account implements bank.Account {
   private volatile double balance = 0.0;
   public double getBalance() {
      return balance;
   }
```

Problem 5: Visibility

- **Account**

```
class Account implements bank.Account {
   private String number; // unique account number
   private String owner;
   private boolean active;

   public String getNumber() { return number; }
   public String getOwner() { return owner; }
   public boolean isActive() { return active; }
   synchronized setInactive() { active = false; }
   ...
```

  - Is the owner and number which has been set in the constructor visible by the calling thread?
  - Does method isActive return the correct value?

Problem 5: Visibility: Solutions

- **Account**

```
class Account implements bank.Account {
   private final String number;
   private final String owner;
   private volatile boolean active;

   public String getNumber() { return number; }
   public String getOwner() { return owner; }
   public boolean isActive() { return active; }
   synchronized setInactive() { active = false; }
   ...
```

  - Initialized values of owner and number are visible by any calling thread due to the final modifier
  - isActive return the correct value due to the volatile modifier
    - synchronized setInactive() alone does NOT solve the visibility issue

Visibility and Publication Mechanism

- **Publication of Bank instance**

```
public class LocalDriver implements BankDriver {
   private LocalBank bank = null;

   public void connect(String[] args){ bank = new LocalBank(); }
   public Bank getBank(){ return bank; }

   public static class LocalBank implements Bank {
      private final Map<String, LocalAccount> activeAccounts =
          new ConcurrentHashMap<>();
...}
```

  - If the bank reference is declared volatile, the reference `activeAccounts` needs not be declared as final
    - **However it is recommended to declare immutables as final because it allows for local reasoning**

Problem 6: Transfer (naïve approach)

```
public void transfer(Account from, Account to, double amount)
                    throws InactiveException, OverdrawException {

   from.withdraw(amount); // throws exception if overdrawn
                          // or inactive
   to.deposit(amount);
}
```

- **Problem**
  - Account *to* must be active, otherwise amount cannot be deposited

Problem 6: Transfer (Variant 1)

```
public void transfer(Account from, Account to, double amount)
           throws InactiveException, OverdrawException {

   if( !to.isActive() ) throw new InactiveException();

   from.withdraw(amount); // may throw an InactiveException
                          // or an OverdrawException
   to.deposit(amount);
}
```

- **Problem**
  - Account *to* may be removed after evaluation of *to.isActive()*
  - Classical instance of a "check then act" situation

Problem 6: Transfer (Variant 2)

```
public void transfer(Account from, Account to, double amount)
                    throws InactiveException, OverdrawException {

   from.withdraw(amount); // may throw an OverdrawException
                          // or an InactiveException
   try {
      to.deposit(amount);
   } catch(InactiveException e) { // => abort, rollback
      from.deposit(amount); throw e;
   }
}
```

- **Problem**
  - Account *from* may be removed during transfer

Problem 6: Solution: transfer synchronized

```
public synchronized void transfer
                    (Account from, Account to, double amount)
                    throws InactiveException, OverdrawException {

   if( !to.isActive() ) throw new InactiveException();

   from.withdraw(amount); // throws exception if overdraft
                          // or if !from.isActive()
   to.deposit(amount);
}
```

- **Problem**
  - Long queue in front of transfer counter
  - to-Account may still be removed during transfer (unless method closeAccount is synchronized on the same lock as method transfer)

Problem 6: Solution

1) Hold lock on to account in transfer

```
public void transfer(Account from, Account to, double amount)
          throws InactiveException, OverdrawException {

  synchronized(to) {
      if( !to.isActive() ) throw new InactiveException();
      from.withdraw(amount); // throws exception if overdrawn
                             // or if inactive
      to.deposit(amount);
   }
}
```

Problem 6: Solution (cont.)

1) Or hold lock on from account in transfer

```
public void transfer(Account from, Account to, double amount)
                throws InactiveException, OverdrawException {

   synchronized(from) {
      from.withdraw(amount); // throws exception if overdrawn
                             // or if inactive
      try{
         to.deposit(amount); // exception if !to.isActive()
      }
      catch(InactiveException e){ // => abort, rollback
         from.deposit(amount);    // will succeed
         throw e;
      }
   }
}
```

Problem 6: Solution (cont.)

2) Protect *closeAccount with the same lock*

3) Check then act => same lock for deposit and withdraw

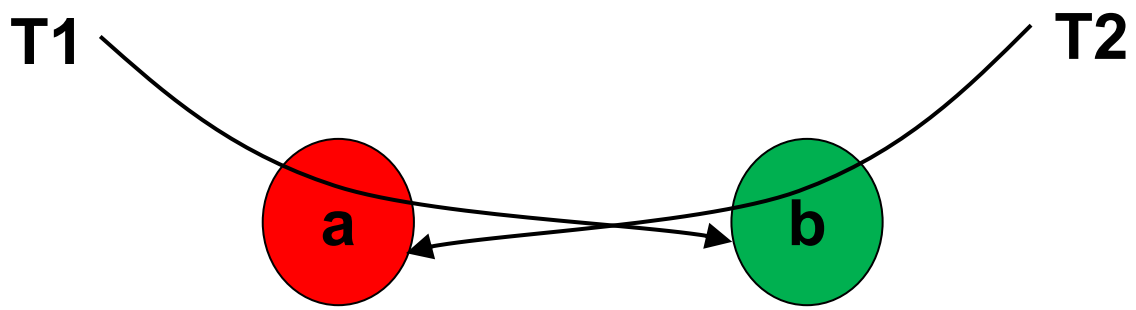
```
public boolean closeAccount(String number){
   Account a = accounts.get(number);
   if(a != null) {
      synchronized(a) {
         if(!a.active) return false;
         if(a.getBalance() == 0) a.active = false;
         return true;
      }
   }
   return false;
}

class Account {
   volatile boolean active = true;
   ...
```

Problem 6: Solution (cont.)

- **Problem: transfer(a,b) | transfer(b,a) may lead to a deadlock**

```
transfer(a, b, amount) {
  synchronized(a) {
    if(!a.isActive()) throw ...
    b.withdraw(amount);
    ...
  }
}
```

```
transfer(b, a, amount) {
  synchronized(b) {
    if(!b.isActive()) throw ...
    a.withdraw(amount);
    ...
  }
}
```



Problem 6: Transfer: Locking two accounts

- **Solution: ordered access to locks**

```
public void transfer(Account from, Account to, double val)
                throws InactiveException, OverdrawException {

   Account first, second;
   if(from.getNumber().compareTo(to.getNumber())<0) {
      first = from; second = to;
   }
   else {
      first = to; second = from;
   }
   synchronized(first) {
      synchronized(second) {
         // do something with from and to
      }
   }
}
```

Common Mistakes / Lessons Learned

- **Problem: Check then act sequences are not atomic**
  - Situation may change right after check due to concurrent activity
    - "check then act" - sequences need to be guarded by a lock
  - If variable is part of an invariant which contains other variables as well: Synchronize EVERY access to the related variables with the same lock
    - Example INV: `active == false => balance == 0`
    - Including read access (=> visibility)
- **Problem: No visibility guarantees if you do nothing**
  - If variable does not change: final
  - If variable is mutable: volatile
  - If variable is mutable and if the new value depends on previous values or if several variables have to be updated: synchronized
- **Writing thread-safe code is challenging!**