Immutability and Threading

Immutable Objects

- Objects that never change their states.
  - Only getter methods; no setter methods.
- No need to worry about race conditions.
  - It is always thread-safe.
  - No locking; no performance loss.
- An example: `java.lang.String`
  - ```
    char[] str = {'u', 'm', 'b'};
    String string = new String(str);
    ```
  - `String string = "umb";`
  - A series of constructors to initialize string data.
  - All non-constructor methods never change the initialized string data.

Methods in `String`

```
• String str = "umb";
                       // Create a String instance that contains "umb"
                       // and assigns it to str.
• System.out.println( str );            // umb
• System.out.println( str.length() );   // 3
• System.out.println( str );            // umb

• System.out.println( str.replace("b","l"));// uml
                       // replace() creates a new String instance that
                       // contains "uml" and returns it.
• System.out.println( str );            // umb
                       // str still contains "umb"

• System.out.println( str.toUpperCase() );  // UMB
                       // toUpperCase() creates a new String instance
                       // that contains "UMB" and returns it.

• System.out.println( str );            // umb
                       // str still contains "umb"
```



```
• String str = "umb";
             // Create a String instance that contains "umb"
             // and assigns it to str.
• System.out.println( str );        // umb

• str = "uml";
             // Create another String instance that contains "uml"
             // and assigns it to str.
• System.out.println( str );        // uml
```
```

`String`

- Maintains the initialized string data (e.g. “umb”) in its *private* and *final* variable:
  - `private final char value[];`
  - Once a value is assigned to a final variable, the value cannot be changed afterward.
    - No methods of `String` can change the value.
- Is defined as a *final* class.
  - A final class cannot be extended (sub-classed).
  - Prevents its sub-classes from updating the initialized string data.

Fragile Immutable Class

```
class Str{
    protected String data;

    StrData(String data){ this.data = data; }
    public String getStr(){ return data; }    }
```

```
class Str2 extends Str{
    ChildStrData(String data){ super(data); }
    public void updateStr(String newStr){
        data = newStr; }    }
```

- `Str` is immutable itself, but it is fragile.
  - Immutable: It has no public setter methods.
  - Fragile: It allows its subclasses to be mutable (i.e. to have public setter methods).
  - Should have been a final class that has a private and final data field.

Note that…

- Whenever appropriate, use both *final* and *private* as often as possible in multi-threaded programs.
  - A small change can turn thread-safe code to be thread-unsafe.
- Clearly state immutability in program comments, API documents, design documents, etc.
  - Use {frozen} or {immutable} in UML class diagrams

Other Immutable Classes

- Wrapper classes for primitive types
  - java.lang.Boolean for boolean
  - java.lang.Byte for byte
  - java.lang.Character for char
  - java.lang.Double for double
  - java.lang.Float for float
  - java.lang.Integer for int
  - java.lang.Long for long
  - java.lang.Short for short
  - …etc.

Immutable Classes are Good for…

- For API designers
  - Immutable classes never require locking (thread synchronization) for read operations.
  - They are free from concurrency issues.
    - Makes it easier to do debugging.

- for API users
  - Immutable classes are free from performance loss due to locking.

Well, Not All Classes can be Immutable…

- Immutable classes are useful.
- However, in practice, most classes are (or must be) mutable.

- Think of separating a class to mutable and immutable parts
  - if read operations are called very often.

An Example: `String` and `StringBuilder`

- Both represent string data.

- `String`
  - Immutable: Its state never change.
  - Thread-safe
  - Faster (or equally fast) to perform read operations.

- `StringBuilder`
  - Mutable: Its state can change through its public methods.
  - Not thread-safe
  - A LOT faster to perform write operations.


- ```
  String str = "UMass";
  str = str + " Boston";          // "UMass Boston"
      // String concatenation. Automatically transformed to:
      // str = new StringBuilder().append(str1).append(str2).toString();
      // Creates two instances and calls two methods.
  ```

- ```
  StringBuilder sbuilder(str);
  sbuilder.append(" Boston);
  str = sbuilder.toString();
  ```

- Write operations (incl. `append()`, `replace()`, etc.) can run faster with `StringBuilder`.

- ```
  ArrayList<String> emailAddrs = ...;
  String commaSeparatedEmailAddrs;
  for(emailAddr: emailAddrs){
      commaSeparatedEmailAddrs += emailAddr + ", ";  }
  ```

- ```
  StringBuilder commaSeparatedEmailAddrs;
  for(emailAddr: emailAddrs){
      commaSeparatedEmailAddrs.append(emailAddr).append(", ";  }
  ```

- The latter code can run 20-100% faster depending on the number of collection elements.

- Use `String` (immutable class) for read operations
- Use `StringBuilder` (mutable class) for write operations

- `String`-to-`StringBuilder` conversion is implemented in a constructor of `StringBuilder`.

- `StringBuilder`-to-`String` conversion is implemented in a constructor of `String`.

`StringBuffer`

- Provides the same set of public methods as `StringBuilder` does.

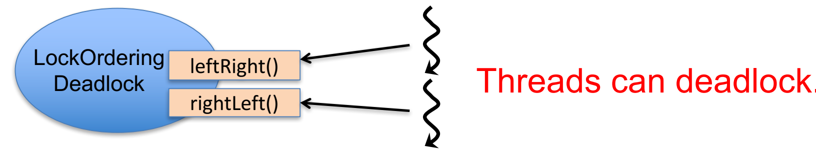
- `StringBuffer` (since Java 1.0)
  - All public methods are thread-safe with locking.
  - Client code of `StringBuffer` may still require locking.

- `StringBuilder` (since Java 5)
  - All public methods are NOT thread-safe.
  - Client code of `StringBuilder` require locking.

User-defined Immutable Class

Lock-ordering Deadlocks

Lock-ordering Deadlocks




```
• class LockOrderingDeadlock{
   private ReentrantLock left = new ReentrantLock();
   private ReentrantLock right = new ReentrantLock();

   public void leftRight(){
       left.lock();
       right.lock();
       // atomic code
       right.unlock();
       left.unlock();
   }

   public void rightLeft(){
       right.lock();
       left.lock();
       // atomic code
       left.unlock();
       right.unlock();
   }
```

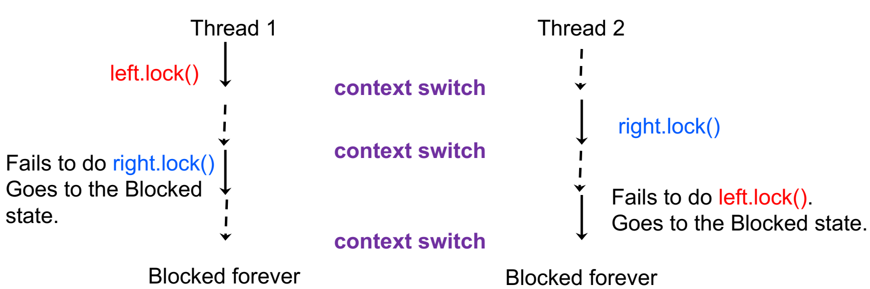
```
• class LockOrderingDeadlock{
   private ReentrantLock left = new ReentrantLock();
   private ReentrantLock right = new ReentrantLock();

   public void leftRight(){
       left.lock();
       right.lock();
       // atomic code
       right.unlock();
       left.unlock(); }

   public void rightLeft(){
       right.lock();
       left.lock();
       // atomic code
       left.unlock();
       right.unlock(); }
```

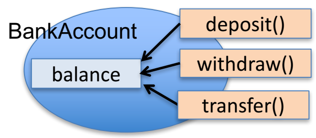
A context switch can occur here.

A context switch can occur here.



- Problem:
  - Threads try to acquire *the same set of locks* in *different orders*.
    - *Inconsistent* lock ordering
      - Thread 1: left → right
      - Thread 2: right → left
- To-do:
  - Have all threads acquire the locks in a *globally-fixed* order.
- **Be careful when you use multiple locks in order!**

Dynamic Lock-ordering Deadlocks

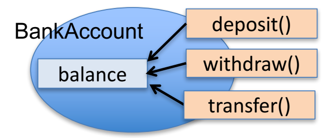



```
class BankAccount{
    private ReentrantLock lock;
    ...
    public void deposit(...)
    public void withdraw(...)
    public void transfer(...)
}
```

```
void deposit(double amount){
    lock.lock();
    balance += amount;
    lock.unlock(); }
```

```
void withdraw(double amount){
    lock.lock();
    balance -= amount;
    lock.unlock(); }
```

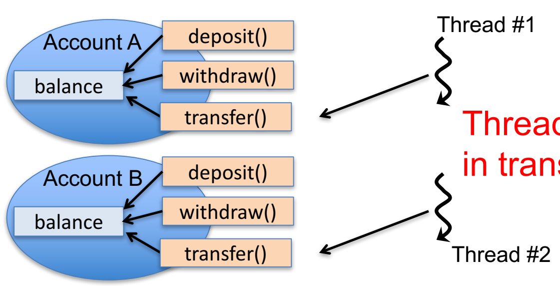
```
void void transfer(Account destination, double amount){
    lock.lock();
    if( balance < amount )
        // generate an error msg or throw an exception
    else{
        withdraw(amount);              // Nested locking. No problem.
        destination.deposit(amount);   // Acquire another lock.
    }
    lock.unlock();}
```




```
class BankAccount{
    private ReentrantLock lock;
    ...
    public void deposit(...)
    public void withdraw(...)
    public void transfer(...)
}
```

```
void transfer(Account destination, double amount){
    lock.lock();
    if( balance < amount )
        // generate an error msg or throw an exception
    else{
        withdraw(amount);              // Nested locking. No problem.
        destination.deposit(amount);   // Acquire another lock.
    }
    lock.unlock();}
```

- It looks as if all threads acquire the two locks (this.lock and destination.lock) in the same order.
- However, this code can have a lock-ordering deadlock.

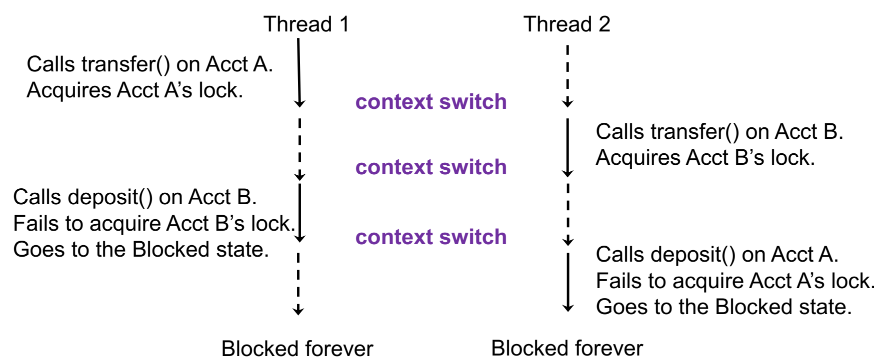


Threads can deadlock in transfer().

- Imagine a scenario where
  - a thread (#1) transfers money from Account A to B
  - another thread (#2) transfers money from B to A.

```
public void transfer(Account destination, double amount){
    lock.lock();
    if( balance < amount )
        // generate an error msg or...
    else{
        withdraw(amount);
        destination.deposit(amount);
    }
    lock.unlock();}
```

A context switch can occur here.



- Problem
  - Threads try to acquire *the same set of locks* in *different orders*.
    - *Inconsistent* lock ordering.
      - Thread 1: Acct A's lock → Acct B's lock
      - Thread 2: Acct B's lock → Acct A's lock
  - This can occur with bad timing although code looks OK.
    - A -> B and C -> D at the same time (No lock-ordering deadlock)
    - A -> B and B -> A at the same time (Possible lock-ordering deadlock)
- **Be careful when you use multiple locks in order!!!**

Solutions

- Static lock
- Timed locking
- Ordered locking
- Nested tryLock()

Solution 1: Static Lock

```
• private static ReentrantLock lock = new ReentrantLock();

• public void deposit(double amount){
      lock.lock();
      this.balance += amount;
      lock.unlock(); }

• public void transfer(Account destination, double amount){
      lock.lock();
      if( this.balance < amount )
         // generate an error msg or throw an exception
      else{
         this.withdraw(amount);        // Nested locking
         destination.deposit(amount);  // Nested locking!
      }
      lock.unlock();}
```

- Pros
  - Simple solution
- Cons
  - Performance penalty
    - Transfers on different accounts are performed sequentially (not concurrently).
    - Deposit operations on different accounts are performed sequentially (not concurrently).
    - Withdrawal operations on different accounts are performed sequentially (not concurrently).

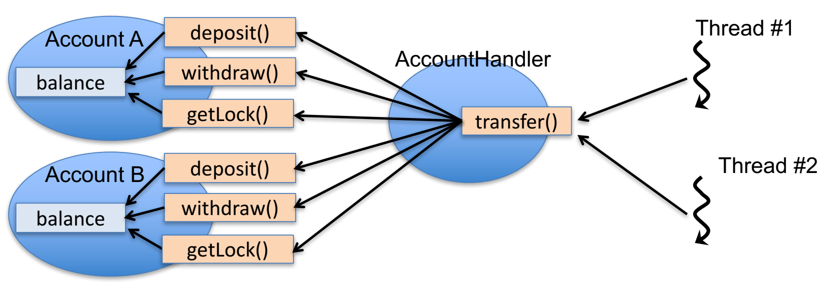
Solution 2: Timed Locking

```
• public void deposit(double amount){
      if( !lock.tryLock(3, TimeUnit.SECONDS) ){
          // generate an error msg or throw an exception
      }else{
          this.balance += amount;
          lock.unlock();} }

• public void transfer(Account destination, double amount){
      lock.lock();
      if( this.balance < amount )
          // generate an error msg or throw an exception
      else{
          this.withdraw(amount);       // Nested locking
          destination.deposit(amount);
      }
      lock.unlock();} //Make sure to release this lock when
                      // an error/exception occurs.
```

- Pros
  - Simple solution
  - More efficient than Solution #1
    - By using a non-static lock
- Cons
  - Transfers and deposits might never be completed.
    - May look like unprofessional.

Solution 3: Ordered Locking




```
• public void transfer( Account source,
                        Account destination,
                        double amount){
      if( source.getAcctNum() < destination.getAcctNum() ){
          source.getLock().lock();
          destination.getLock().lock();
          if( source.getBalance() < amount )
              // generate an error msg or throw an exception
          else{
              source.withdraw(amount);     //Nested locking
              destination.deposit(amount);//Nested locking
          }
          destination.getLock().unlock();
          source.getLock().unlock();
      }
      else if( source.getAcctNum() > destination.getAcctNum() ){
          destination.getLock().lock();
          source.getLock().lock();
          ...
          source.getLock().unlock();
          destination.getLock().unlock();
      }
```

- Pros
  - Locks are always acquired in the same order.
  - More efficient than Solution #1
    - By using a non-static lock
  - More professional than Solution #2
    - Transfers and deposits complete for sure.
- Cons
  - Using an application-specific/dependent data.
  - Account numbers should not be changed after accounts are set up.
    - If you allow dynamic changes of account numbers, you need to use an extra lock.

Solution 3a: Ordered Locking with Instance IDs

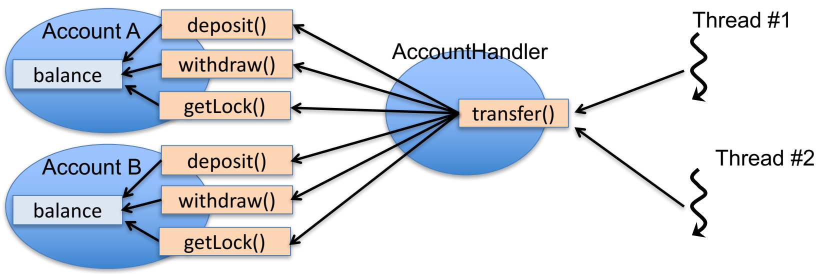
- Instance IDs
  - Unique IDs (hash code) that the local JVM assigns to individual class instances.
    - Unique and intact on the same JVM
      - 2 instances of the same class have different IDs.
      - No instances share the same ID.
      - IDs never change after they are assigned to instances.
  - Use System.identifyHashCode()

```
public void transfer( Account source,
                      Account destination,
                      double amount){
    int sourceID = System.identifyHashCode(source);
    int destID = System.identifyHashCode(destination);

    if( sourceID < destID ){
        source.getLock().lock();
        destination.getLock().lock();
        if( source.getBalance() < amount )
            // generate an error msg or throw an exception
        else{
            source.withdraw(amount);      //Nested locking
            destination.deposit(amount);//Nested locking
        }
        destination.getLock().unlock();
        source.getLock().unlock();
    }
    if( sourceID > destID ){
        destination.getLock().lock();
        source.getLock().lock();
        ...
        source.getLock().unlock();
        destination.getLock().unlock();
    }
```

- Pros
  - Locks are always acquired in the same order.
  - More efficient than Solution #1
    - By using a non-static lock
  - More professional than Solution #2
    - Transfers and deposits complete for sure.
  - No application-specific data (e.g., account numbers) are necessary to order locking.
- Cons
  - N/A

Solution 4: Nested Timed Locking



- Use nested tryLock() calls to implement an ALL-OR-NOTHING policy.
  - Acquire both of A's and B's locks, OR
  - Acquire none of them.
- Avoid a situation where a thread acquires one of the two locks and fails to acquire the other.

-
```
public void transfer(Account source,
                     Account destination,
                     double amount){
 Random random = new Random();

 while(true){
     if( source.getLock().tryLock() ){
         try{
             if( destination.getLock().tryLock() ){
                 try{
                     if( source.getBalance() < amount )
                         // generate an error msg/exception
                     else{
                         source.withdraw(amount);
                         destination.deposit(amount);
                     }
                     return;
                 }finally{
                     destination.getLock().unlock();
                 }
             }
         }finally{
             source.getLock().unlock();
         }
     }
     Thread.sleep(random.nextInt(1000));
 }
}
```

- If the first tryLock() fails, then sleep.
- If the first tryLock() succeeds but the second one fails, unlock the first lock and sleep.

- Pros
  - More efficient than Solution #1
    - By using a non-static lock
  - More professional than Solution #2
    - Transfers and deposits complete for sure.
  - No application-specific data (e.g., account numbers) are necessary to order locking.
- Cons
  - Not that simple