Another Warning Sign for Deadlocks: Alien Method Calls

Alien Method Call

```
class A{
  private B b;
  private ReentrantLock lock;

  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock(); } }
```

```
Class B{
  private A a;

  public void b1(){
    do something;
  }
}
```

- This code is deadlock-prone.
  - Calling an *alien method* with a lock held.
    - Alien method: method in another class and overridable method in the same class

Alien Method Call

```
class A{
  private B b;
  private ReentrantLock lock;

  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock(); } }
```

```
Class B{
  private A a;

  public void b1(){
    do something;
  }
}
```

- This code is deadlock-prone.
  - Calling an *alien method* with a lock held.
    - Alien method: method in another class and overridable method in the same class
- It can cause a deadlock if an alien method (b1())…
  - Runs an infinite loop.
  - Tries to acquire B's lock and A's lock.
    - Can cause a lock-ordering deadlock
  - Spawns a new thread and does a callback.

Lock-ordering Deadlock

```
class A{
  private B b;
  private ReentrantLock lock;

  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock(); }

  public void a2(){
    lock.lock();
    b.b1();
    lock.unlock(); }
}
```

```
Class B{
  private A a;
  private ReentrantLock lock;

  public void b1(){
    lock.lock();
    a.a2();
    lock.unlock(); }
}
```

- This code is deadlock-prone.
- It can cause a lock-ordering deadlock if an alien method (b1()) tries to acquire B's lock and then A's lock.

```
class A{
  private B b;
  private ReentrantLock lock;
  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock();
  }

  public void a2(){
    lock.lock();
    b.b1();
    lock.unlock(); }
}
```

```
Class B{
  private A a;
  private ReentrantLock lock;
  public void b1(){
    lock.lock();
    a.a2();
    lock.unlock();
  }
}
```

Thread #1

Thread #2

- Thread #1 acquires A's lock, B's lock and A's lock.
  - No problem to acquire A's lock twice (nested locking)
- Thread #2 acquires B's lock and A's lock.

Deadlock by a Concurrent Callback

```
class A{
  private B b;
  private ReentrantLock lock;

  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock(); }

  public void a2(){
    lock.lock();
    ...
    lock.unlock(); }
}
```

```
Class B{
  private A a;

  public void b1(){
    ...
    Thread th = new Thread(
      ()->a.a2() );
    th.start();
    ...
    th.join();
  }
}
```

Important Note

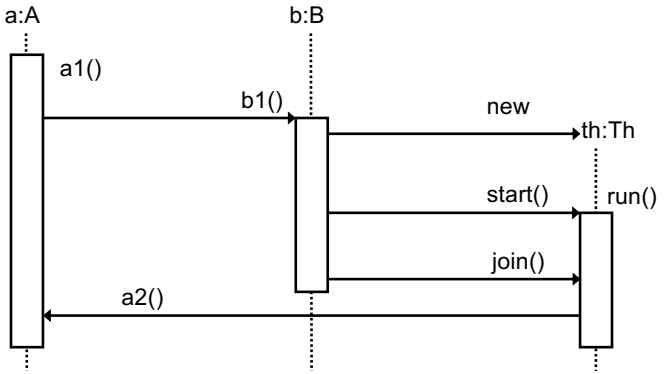
```
class A{
  private B b;
  private ReentrantLock lock;

  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock(); }
}
```

```
Class B{
  private A a;

  public void b1(){...}
}
```

- If you implement A and B AND use them by yourself,
  - Be careful NOT to cause a deadlock.
- If you implement A and B as an API designer and leave B's implementation to others,
  - you have NO WAYS to prevent them from causing lock-ordering deadlocks.

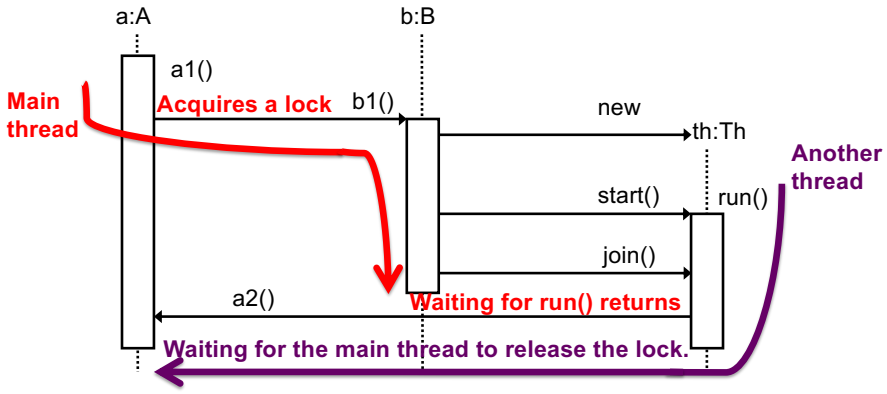
```
class A{
  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock(); }
  public void a2(){
    lock.lock();
    ……
    lock.unlock(); } }
```

```
class B{
  public void b1(){
    Thread Th = new Thread(
      ()->a.a2() );
    th.start();
    ...
    th.join(); } }
```



```
class A{
  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock(); }
  public void a2(){
    lock.lock();
    ……
    lock.unlock(); } }
```

```
class B{
  public void b1(){
    Thread Th = new Thread(
      ()->a.a2() );
    th.start();
    ...
    th.join(); } }
```



Important Note

```
class A{
  private B b;
  private ReentrantLock lock;

  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock(); }
}
```

```
Class B{
  private A a;

  public void b1(){
    do something;
  }
}
```

- If you implement A and B AND use them by yourself,
  - Be careful NOT to cause a deadlock.
- If you implement A and B as an API designer and leave B's implementation to others,
  - you have NO WAYS to prevent them from causing lock-ordering deadlocks.

Another Deadlock Case

```
class A{
  private ReentrantLock lock;

  public void a1(){
    lock.lock();
    a2();
    lock.unlock();
  }
  protected void a2(){}
}
```

- a2() is an alien method. Deadlock prone.
  - Alien method: method in another class and overridable method in the same class
- If you implement a1() and leave a2()'s implementation to others,
  - you have NO WAYS to prevent them from causing a deadlock.

To Eliminate Potential Deadlocks…

```
class A{
  private B b;
  private ReentrantLock lock;

  public void a1(){
    lock.lock();
    ...
    lock.unlock();
    b.b1(); // open call }
}
```

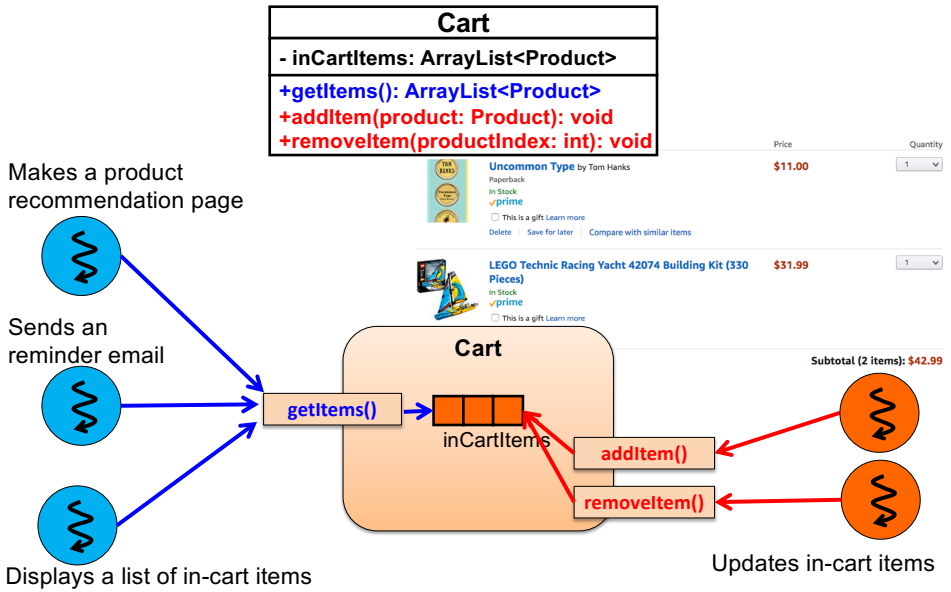
```
Class B{
  private A a;

  public void b1(){
    do something;
  }
}
```

- Open call
  - Avoid calling an alien method (b1()) with a lock held (i.e., in atomic code)
  - Instead, move the method call outside the atomic code.

Rule of Thumb

- Try NOT to call an alien method from atomic code.
- Call an alien method outside atomic code.
  – Open call.
- Carefully eliminate race conditions.
- If necessary, compromise to favor being deadlock-free than being free from race conditions.

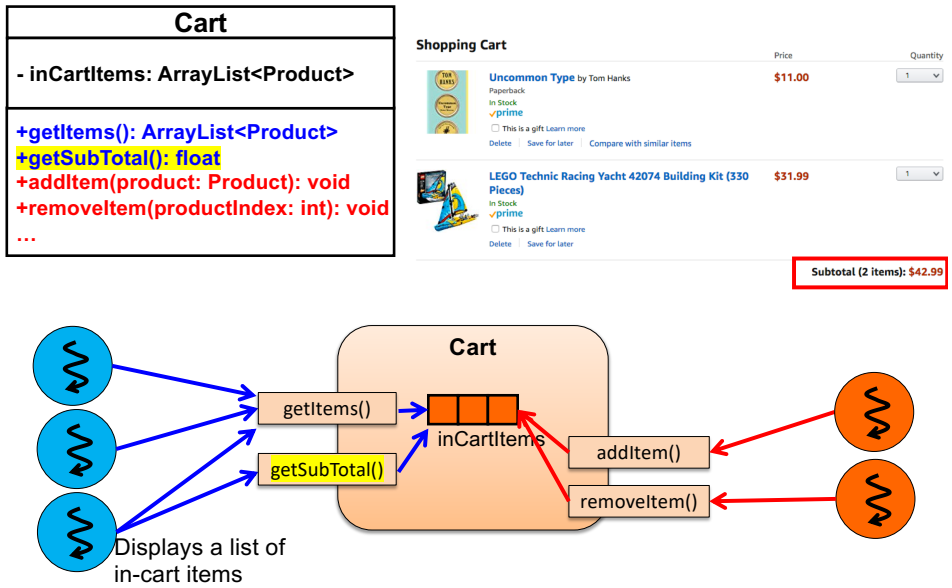
Exercise: Online Shopping Cart



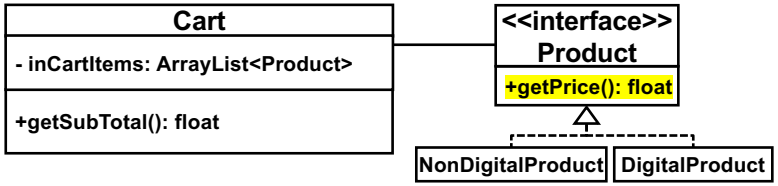
Recap: Thread-safe Shopping Cart

```
class Cart{
 private ArrayList<Product> inCartItems;     // ArrayList is not thread-safe
 private ReentrantLock lock = new ...;

 public ArrayList<Product> getItems(){
   lock.lock();
   return inCartItems;                 // READ
   lock.unlock(); }

public void addItem(Product item){
   lock.lock();
   inCartItems.add(item);              // WRITE
   lock.unlock(); }

 public void removeItem(int productIndex){
   lock.lock();
   inCarItems.remove(productIndex); // WRITE
   lock.unlock(); } }
```

A New Method in Cart



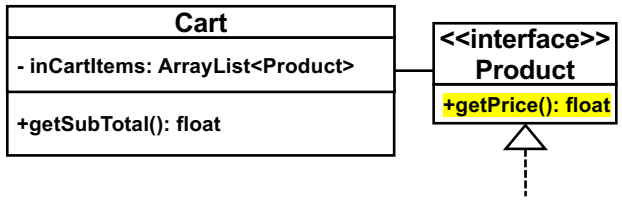
```
class Cart{
 private ArrayList<Product> inCartItems;     // ArrayList is not thread-safe
 private ReentrantLock lock = new ...;

 public ArrayList<Product> getItems(){
   lock.lock();
   return inCartItems;                 // READ
   lock.unlock(); }

public void addItem(Product item){
   lock.lock();
   inCartItems.add(item);              // WRITE
   lock.unlock(); }

 public void removeItem(int productIndex){
   lock.lock();
   inCarItems.remove(productIndex); // WRITE
   lock.unlock(); }

 public float getSubTotal(){                 // New method
   float subtotal;                           // Local variable; NOT shared by
   lock.lock();                              // multiple threads
   for(Product item: inCarItems){    // READ
     subtotal += item.getPrice(); }
   lock.unlock();
   return subtotal; } }
```




```
class Cart{
 private ArrayList<Product> inCartItems;
 private ReentrantLock lock = new ...;

 public float getSubTotal(){
   float subtotal;
   lock.lock();
   for(Product item: inCarItems){
     subtotal += item.getPrice(); }
   lock.unlock();
   return subtotal; } }
```

```
class NonDigitalProduct
    implements Product {
 private float price;
 private ReentrantLock lockP = ...;

 public float getPrice(){
   lockP.lock();
   return price;
   lockP.unlock(); }
```

- **`getPrice()`** may be implemented this way.
  - Two locks (**`lock`** and **`lockP`**) are acquired in order.
  - Very risky. Try to do open call; move **`getPrice()`** outside the atomic code.




```
class Cart{
 private ArrayList<Product> inCartItems;
 private ReentrantLock lock = new ...;

 public float getSubTotal(){
   float subtotal;
   lock.lock();
   for(Product item: inCarItems){    // READ
     subtotal += item.getPrice(); }
   lock.unlock();
   return subtotal; } }
```

- **`getPrice()`** is an alien method for Cart.
  - **`getPrice()`** is called with a **`lock`** held.
  - You can do that, if you can fully control how it is implemented.
  - Otherwise, this code is risky/deadlock-prone.

```
class Cart{
 private ArrayList<Product> inCartItems;
 private ReentrantLock lock = ...;

 public float getSubTotal(){
   float subtotal;
   ArrayList<Product> inCarItemsLocal;

   lock.lock();
   inCarItemsLocal =
     new ArrayList(inCarItems);
   lock.unlock();

   for(Product item: inCarItemsLocal){
     subtotal += item.getPrice(); } // OPEN CALL }}
```

```
class NonDigitalProduct
    implements Product {
 private float price;
 private ReentrantLock lockP = ...;

 public float getPrice(){
   lockP.lock();
   return price;
   lockP.unlock(); }
```

- First, copy in-cart items in **`inCartItems`** (data field) to **`inCartItemsLocal`** (local variable).
  - Guard **`inCartItems`** (shared variable) with **`lock`**. Otherwise, race conditions can occur.
- Then, call the alien method **`getPrice()`** without holding **`lock`**.
  - A local variable (**`inCartItemsLocal`**) is never shared by multiple threads.
    - It is created for each thread. No need to guard it with a lock.

Note: Potential Race Conditions

```
class Cart{
 private ArrayList<Product> inCartItems;
 private ReentrantLock lock = ...;

 public float getSubTotal(){
   float subtotal;
   ArrayList<Product> inCarItemsLocal;

   lock.lock();
   inCarItemsLocal =
     new ArrayList(inCarItems);
   lock.unlock();

   for(Product item: inCarItemsLocal){
     subtotal += item.getPrice(); }// OPEN CALL }}
```

```
class NonDigitalProduct
    implements Product {
 private float price;
 private ReentrantLock lockP = ...;

 public float getPrice(){
   lockP.lock();
   return price;
   lockP.unlock(); }
```

Context switches can occur. Other threads may add or remove items by calling addItem() or removeItem().

- **`inCartItems`** (data field) may not be in synch with **`inCartItemsLocal`** (local variable).
  - Race conditions!

Rule of Thumb

- Try NOT to call an alien method from atomic code.
- Call an alien method outside atomic code.
  - Open call.
- Carefully eliminate race conditions.
- If necessary, compromise to favor being deadlock-free than being free from race conditions.

Dilemma

- First solution
  - Pros: Race-condition free
    - Read/write operations on the local variable **`subtotal`** is thread-safe.
  - Cons: Deadlock-prone
    - Need to call the alien method **`getPrice()`** with a lock held.
- Second solution
  - Pros: Deadlock-free
    - Open call: **`getPrice()`** is called without a lock held.
  - Cons: Not perfectly race-condition free
    - Read/write operations on the local variable **`subtotal`** is thread-safe.
    - **`inCartItems`** (data field) and **`inCartItemsLocal`** (local variable) may not be in sync.