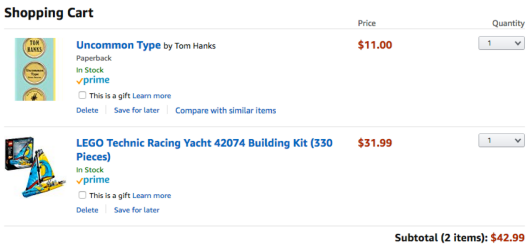
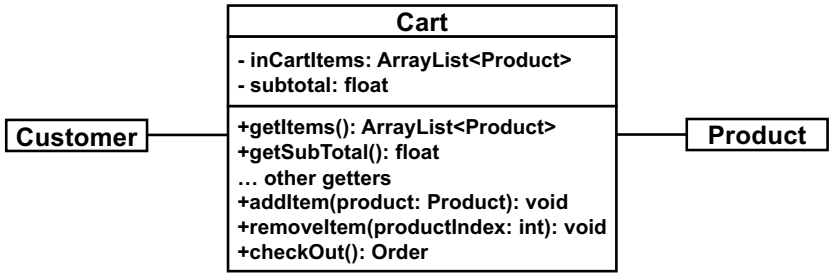
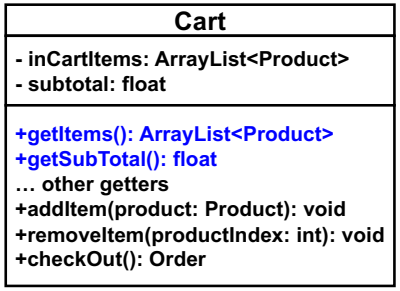
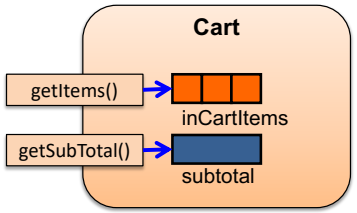
Quiz

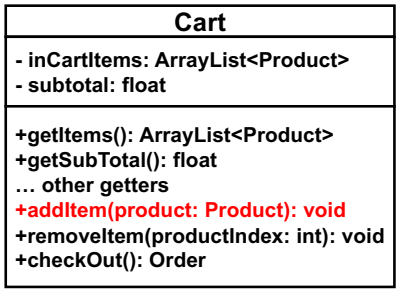
- Suppose you are implementing shopping carts for an online retailer.

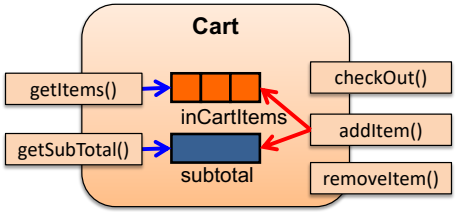


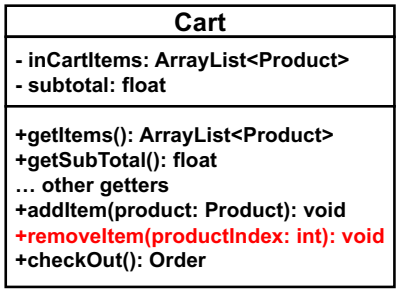


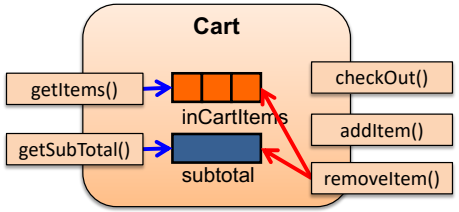








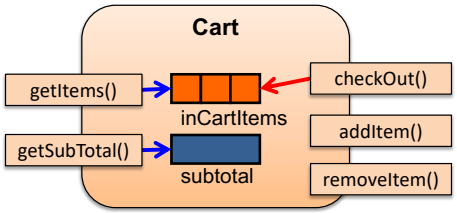


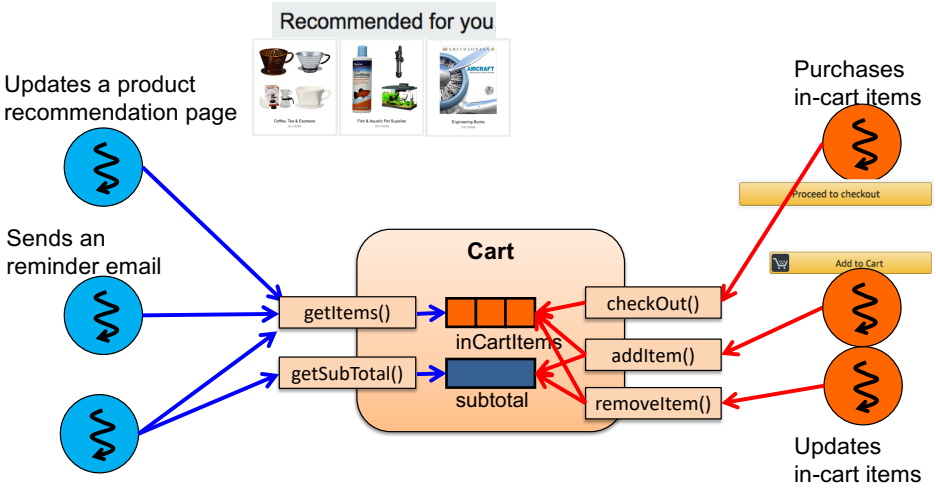


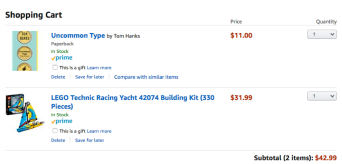
Cart

- inCartItems: ArrayList<Product>
- subtotal: float

+getItems(): ArrayList<Product>
+getSubTotal(): float
… other getters
+addItem(product: Product): void
+removeItem(productIndex: int): void
+checkOut(): Order



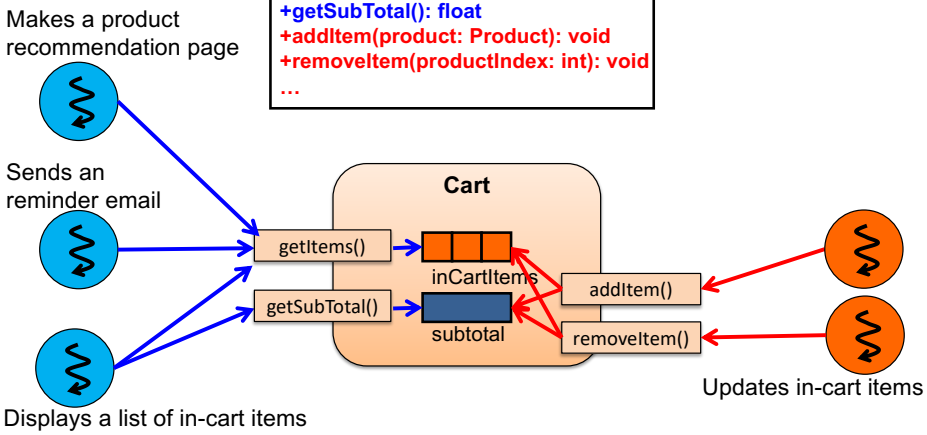


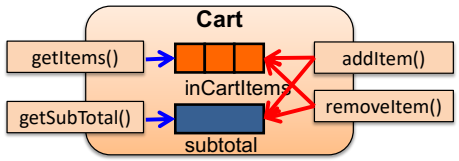


Cart

- inCartItems: ArrayList<Product>
- subtotal: float

+getItems(): ArrayList<Product>
+getSubTotal(): float
+addItem(product: Product): void
+removeItem(productIndex: int): void
…

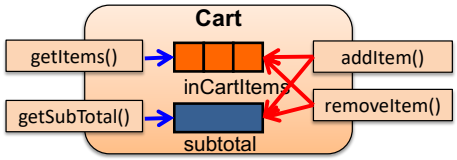





```
class Cart{
 private ArrayList<Product> inCartItems;
 private float subtotal;

 public ArrayList<Product> getItems(){
   return inCartItems; }

 public float getSubTotal(){
   return subtotal; }

 public void addItem(Product item){
   inCartItems.add(item);
   subtotal += item.getPrice(); } }

 public void removeItem(int productIndex){
   subtotal -=
     inCarItems.get(productIndex).getPrice();
   inCarItems.remove(productIndex); } }
```

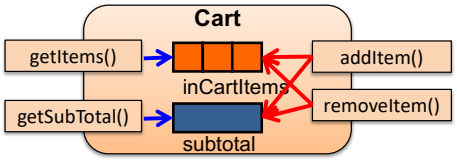
- `inCarItems` and `subtotal` are shared by threads.
- Need to guard them against concurrent accesses.

- **Cart is not thread-safe. Explain why and how to make it thread-safe.**
  - **Note: ArrayList is not thread-safe. (All of its public methods such as add() and remove() are not thread-safe.)**




```
class Cart{
 private ArrayList<Product> inCartItems;
 private float subtotal;

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

 public float getSubTotal(){
   return subtotal; }               // READ

 public void addItem(Product item){
   inCartItems.add(item);           // WRITE
   subtotal += item.getPrice(); } } // WRITE

 public void removeItem(int productIndex){
   subtotal -=
     inCarItems.get(productIndex).getPrice(); // READ, WRITE
   inCarItems.remove(productIndex); } }       // WRITE
```

- `inCarItems` and `subtotal` are shared by threads.
- Need to guard them against concurrent accesses.
  - Identify every single *read* and *write* logic on each shared variable.
  - Surround it with lock() and unlock() on a lock




```
class Cart{
 private ArrayList<Product> inCartItems;
 private float subtotal;

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

 public float getSubTotal(){
   return subtotal; }               // READ

 public void addItem(Product item){
   inCartItems.add(item);           // WRITE
   subtotal += item.getPrice(); } } // WRITE

 public void removeItem(int productIndex){
   subtotal -=
     inCarItems.get(productIndex).getPrice(); // READ, WRITE
   inCarItems.remove(productIndex); } }       // WRITE
```

- `inCarItems` and `subtotal` are shared by threads.
- Need to guard them against concurrent accesses.
  - Identify every single *read* and *write* logic on each shared variable.
  - Surround it with lock() and unlock() on a lock
  - How many locks to use? ONE or TWO???

Solution 1

```
class Cart{
 private ArrayList<Product> inCartItems;
 private float subtotal;
 private ReentrantLock lockI = new ...;
 private ReentrantLock lockST = new ...;

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

 public float getSubTotal(){
   lockST.lock();
   return subtotal;                 // READ
   lockST.unlock(); }

 public void addItem(Product item){ ... }

 public void removeItem(int productIndex){ ... }
}
```

- Use two locks.
  - One lock per shared variable.

Solution 1

```
class Cart{
 private ArrayList<Product> inCartItems;
 private float subtotal;
 private ReentrantLock lockI = new ...;
 private ReentrantLock lockST = new ...;

 public ArrayList<Product> getItems(){ ... }
 public float getSubTotal(){ ... }

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

   lockST.lock();
   subtotal += item.getPrice(); // WRITE
   lockST.unlock(); }

 public void removeItem(int productIndex){ ... }
}
```

- Use two locks.
  - One lock per shared variable.

Solution 1

```
class Cart{
 private ArrayList<Product> inCartItems;
 private float subtotal;
 private ReentrantLock lockI = new ...;
 private ReentrantLock lockST = new ...;

 public ArrayList<Product> getItems(){ ... }
 public float getSubTotal(){ ... }

 public void addItem(Product item){
   lockI.lock();
   inCartItems.add(item);       // WRITE
   lockI.unlock();
                  // RACE CONDITIONS CAN OCCUR IF A CONTEXT SWITCH HAPPENS
                  // AFTER UNLOCK() AND ANOTHER THREAD UPDATES IN-CART ITEMS
   lockST.lock();
   subtotal += item.getPrice(); // WRITE
   lockST.unlock(); }

 public void removeItem(int productIndex){ ... }
}
```

- Use two locks.
  - One lock per shared variable.

- `inCarItems` and `subtotal` get out of synch due to a race condition.
- Need to run the both WRITE operations atomically

Solution 1

```
class Cart{
 private ArrayList<Product> inCartItems;
 private float subtotal;
 private ReentrantLock lockI = new ...;
 private ReentrantLock lockST = new ...;

 public ArrayList<Product> getItems(){ ... }
 public float getSubTotal(){ ... }

 public void addItem(Product item){
   lockI.lock();
   lockST.lock();
   inCartItems.add(item);       // WRITE
   subtotal += item.getPrice(); // WRITE
   lockST.unlock();
   lockI.unlock();
 }

 public void removeItem(int productIndex){ ... }
}
```

- Use two locks.
  - One lock per shared variable.

- Need to run the both WRITE operations atomically
- Run the two operations with two locks held.

Solution 1

```
class Cart{
 private ArrayList<Product> inCartItems;
 private float subtotal;
 private ReentrantLock lockI = new ...;
 private ReentrantLock lockST = new ...;

 public ArrayList<Product> getItems(){ ... }
 public float getSubTotal(){ ... }

 public void addItem(Product item){
   lockI.lock();
   lockST.lock();
   inCartItems.add(item);       // WRITE
   subtotal += item.getPrice(); // WRITE
   lockST.unlock();
   lockI.unlock();
 }

 public void removeItem(int productIndex){ ... }
}
```

- Use two locks.
  - One lock per shared variable.

- Run the two operations with two locks held.
- Be aware that a thread acquires two locks in order! Be **CAREFUL** not to cause lock-ordering deadlocks.

Solution 1

```
class Cart{
 private ArrayList<Product> inCartItems;
 private float subtotal;
 private ReentrantLock lockI = new ...;
 private ReentrantLock lockST = new ...;

 public ArrayList<Product> getItems(){ ... }
 public float getSubTotal(){ ... }

 public void addItem(Product item){ ... }

 public void removeItem(int productIndex){
   subtotal -= inCarItems.get(productIndex).getPrice(); // READ, WRITE
   inCarItems.remove(productIndex); } }                 // WRITE
```

- Use two locks.
  - One lock per shared variable.

- `removeItem()` performs 3 read/write logic:
  - Getting an item from `inCarItems` with `get()`
  - Updating `subtotal`
  - Removing an item from `inCarItems` with `remove()`
- Need to run all of them atomically
  - Otherwise, `inCarItems` and `subtotal` get out of synch due to a race condition.

Solution 1

```
class Cart{
 private ArrayList<Product> inCartItems;
 private float subtotal;
 private ReentrantLock lockI = new ...;
 private ReentrantLock lockST = new ...;

 public void addItem(Product item){
   lockI.lock();
   lockST.lock();
   inCartItems.add(item);          // WRITE
   subtotal += item.getPrice();    // WRITE
   lockST.unlock();
   lockI.unlock(); }

 public void removeItem(int productIndex){
   lockI.lock();
   lockST.lock();
   subtotal -= inCarItems.get(productIndex).getPrice(); // READ, WRITE
   inCarItems.remove(productIndex);                     // WRITE
   lockST.lock();
   lockI.lock(); } }
```

- Use two locks.
  - One lock per shared variable.

- Run all the 3 read/write logic with two locks held.
- Be aware that a thread acquires two locks in order! Make sure to acquire `lockI` and then `lockST`.
  - Otherwise, lock-ordering deadlocks can occur.

Solution 2

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

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

 public float getSubTotal(){
   lock.lock();
   return subtotal;                 // READ
   lock.unlock(); }

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

 public void removeItem(int productIndex){
   lock.lock();
   subtotal -=
     inCarItems.get(productIndex).getPrice();   // READ, WRITE
   inCarItems.remove(productIndex);             // WRITE
   lock.unlock(); } }
```

- Use one lock.

Solution 1 v.s. Solution 2

- Solution 1
  - Straightforward “lock-per-variable” design
  - Need to acquire multiple locks in order
    - Must have all developers in the team be very careful not to cause lock-ordering deadlocks.
- Solution 2
  - One lock to guard two variables
    - Must have all developers in the team know which variables the lock guards.
  - No worries about lock-ordering deadlocks.

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;
  }
}
```

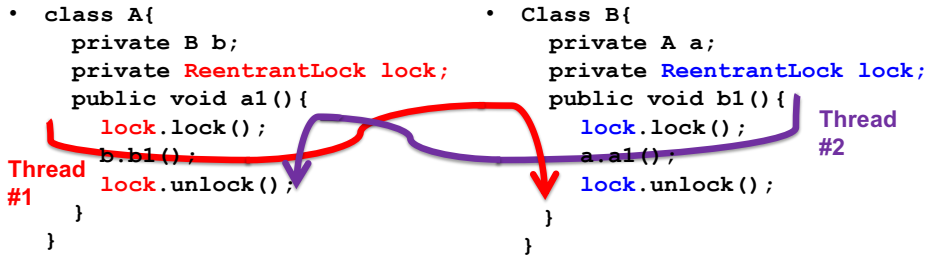
- This code is deadlock-prone.
  - Calling an *alien method* with a lock held.
    - Alien method: method in another class and overridable method in the same class
- It can cause a deadlock if an alien method (b1())…
  - Runs an infinite loop.
  - Acquires B's lock and A's lock.
    - Can cause a lock-ordering 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(); }
}
```

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

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

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



- 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.

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.

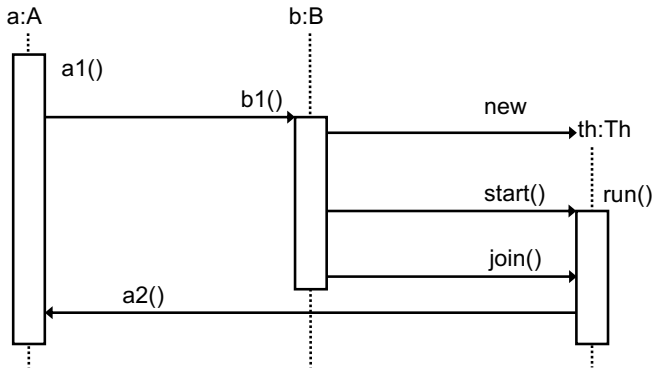
Deadlock by a Concurrent Callback

```
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();
  }
}
```

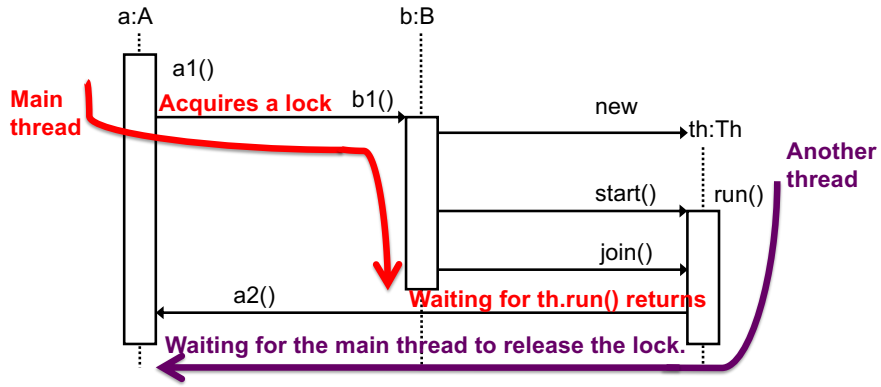
```
class A{
  public void a1(){
    lock.lock();
    b.b1();
    lock.unlock(); }
  public void a2(){
    lock.lock();
    ……
    lock.unlock(); } }
```

```
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 }
}
```

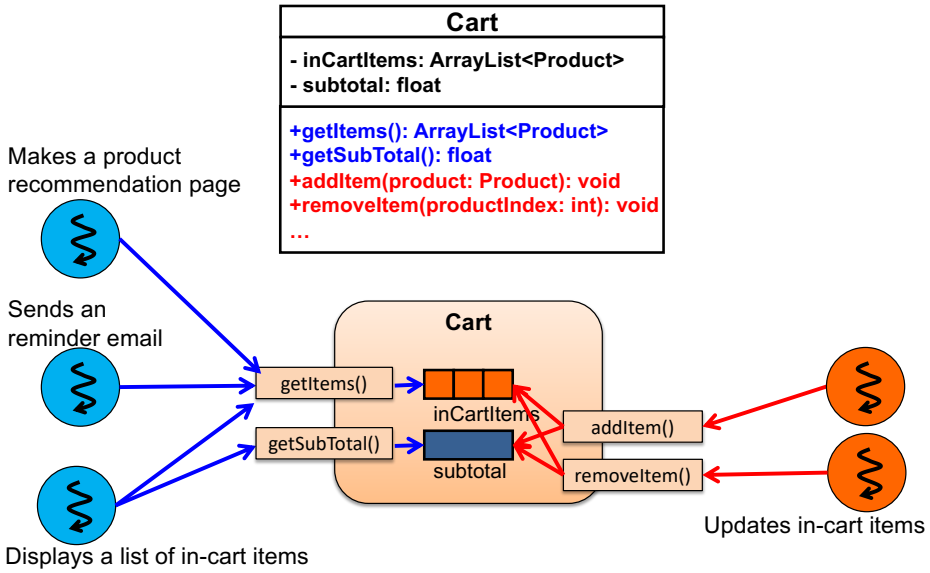
```
Class B{
  private A a;

  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

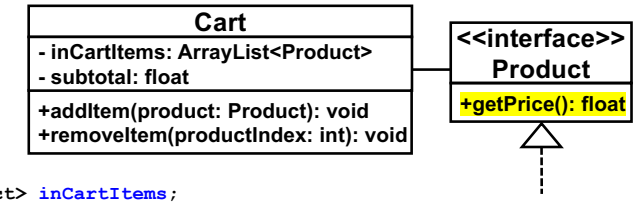
- Try NOT to call an alien method from atomic code.
- Call an alien method outside atomic code.
  - Open call.
- Carefully eliminate race conditions.
- If necessary, compromise to favor being deadlock-free than being free from race conditions.

An Example Open Call



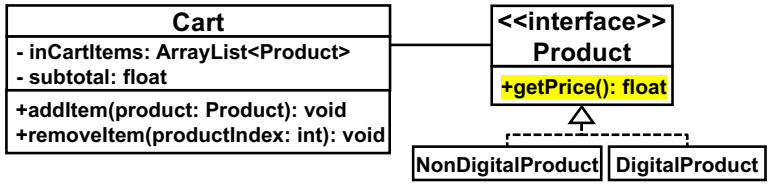

```
class Cart{
 private ArrayList<Product> inCartItems;
 private float subtotal;
 private ReentrantLock lock = new ...;

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

 public float getSubTotal(){
   lock.lock();
   return subtotal;                       // READ
   lock.unlock(); }

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

 public void removeItem(int productIndex){
   lock.lock();
   subtotal -=
     inCarItems.get(productIndex).getPrice();   // READ, WRITE
   inCarItems.remove(productIndex);             // WRITE
   lock.unlock(); } }
```

- Use one lock.
  - Solution 2




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

 public void addItem(Product item){
   lock.lock();
   inCartItems.add(item);
   subtotal += item.getPrice();
   lock.unlock(); }

 public void removeItem(int productIndex){
   lock.lock();
   subtotal -=
     inCarItems.get(productIndex).getPrice();
   inCarItems.remove(productIndex);
   lock.unlock(); } }
```

- `getPrice()` is an alien method for Cart.
  - You can call it in atomic code, if you can fully control how it is implemented.
  - Otherwise, move the method call outside the atomic code.




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

 public void addItem(Product item){
  lock.lock();
  inCartItems.add(item);
  subtotal += item.getPrice();
  lock.unlock(); }

 public void removeItem(int productIndex){
  lock.lock();
  subtotal -=
   inCarItems.get(productIndex).getPrice();
  inCarItems.remove(productIndex);
  lock.unlock(); } }
```

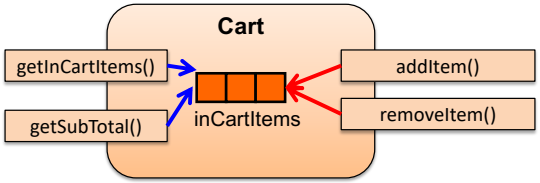
```
class NonDigitalProduct
  implements Product {
 private float price;
 private ReentrantLock lockP = ...;

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

- `getPrice()` may be implemented this way.
  - This code is error-prone/risky.
  - An alien method (`getPrice()`) is called with a `lock` held. Two locks (`lock` and `lockP`) are acquired in order.

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

 public void addItem(Product item){
   lock.lock();
   inCartItems.add(item);
   lock.unlock();
   subtotal += item.getPrice(); } }
   // OPEN CALL!
```

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

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

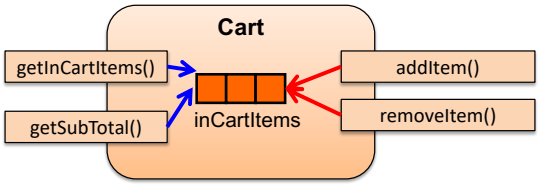
- Do *open call*; call `getPrice()` without holding `lock`.
- Race conditions can occur.
  - `inCartItems` and `subtotal` can be out of synch with each other
- Do we really need `subtotal` as a data field?




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

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

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

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

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

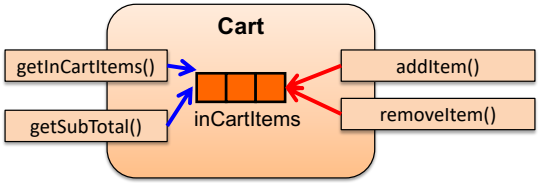
- `removeItem()` does not have to call `getPrice()`.
- Two locks are no longer acquired in order.
  - No need to warry about lock-ordering deadlocks.




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

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

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

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

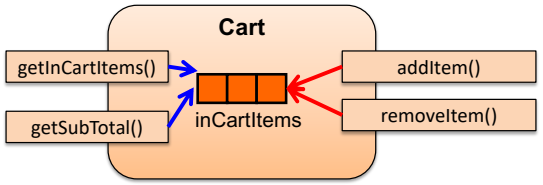
- Remove `subtotal` from `Cart`.
  - No need to have two locks in `Cart`.
- `addItem()` does not have to call `getPrice()`.
- Two locks are no longer acquired in order.
  - No need to warry about lock-ordering deadlocks.




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

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

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

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

- `getSubTotal()` needs to call `getPrice()`.
- It reads a shared variable (`inCartItems`).
  - Need to guard it with `lock`. Otherwise, race conditions can occur.
- A local variable (`subtotal`) is never shared by multiple threads.
  - It is created for each thread. No need to guard it with a lock.




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

 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(); }
```

- This code is error-prone/risky.
  - An alien method (`getPrice()`) is called with a `lock` held. Two locks (`lock` and `lockP`) are acquired in order.
- How should/can we do open call?

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

 public float getSubTotal(){
   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).
  - This requires to read a shared variable (`inCartItems`). Need to guard it with `lock`. Otherwise, race conditions can occur.
- Then, call an 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.

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.