# Race Conditions

CPSC 1181 – O.O.

Jeremy Hilliker

Summer 2017

# Outline

- Race Conditions
  - Scheduling
  - Bank
  - List

- Approaches
  - **Locks**
  - Wrappers
  - Synchronized
  - Active Objects
    - Actor model
  - Atomic operations
  - Ownership
  - Redesign

- Next: Deadlock

# Last time...

- Our threads worked in isolation
- What is they want to share some Objects?

# Problem

- Threads can conflict with each other when they share an object
- Scenario:
  - Thread 1 is updating object X
  - Thread 2 also updates object X
  - One of the changes gets lost
  - Or worse: the object is left in an inconsistent state
    - Some changes from thread 1, but not all
    - Some changes from thread 2, but not all

# Example

- 2 threads updating a bank account
  - Starting balance: $10,000
  - Each thread withdraws $1, 5,000 times
  - Final balance should be $0

```java
public class BankRaceRunnable implements Runnable {

    private final static int NUM_THREADS = 2;
    private final static int INITIAL_BALANCE = 10000;
    private final static int WITHDRAWL_PER_THREAD = INITIAL_BALANCE / NUM_THREADS;
    private final static int EXPECTED_WITHDRAWN = NUM_THREADS * WITHDRAWL_PER_THREAD;

    private static int balance = INITIAL_BALANCE;

    public void run() {
      for(int i = 0; i < WITHDRAWL_PER_THREAD; i++) {
        // balance = balance - 1;
        balance--;
      }
    }

    public static void main(String[] args) throws InterruptedException {
      Thread[] ts = new Thread[NUM_THREADS];
      for(int i = 0; i < NUM_THREADS; i++) {
        ts[i] = new Thread(new BankRaceRunnable());
        ts[i].start();
      }
      for(Thread t : ts) {
        t.join();
      }
```

```
Initial balance: 10000
2 threads withdrawing 5000 each.
10000 expected withdrawn.
Expected ending balance: 0
  Actual ending balance: 4337
```

```
12    public void run() {
13      for(int i = 0; i < WITHDRAWL_PER_THREAD; i++) {
14        // balance = balance - 1;
15        balance--;
16      }
17    }
```

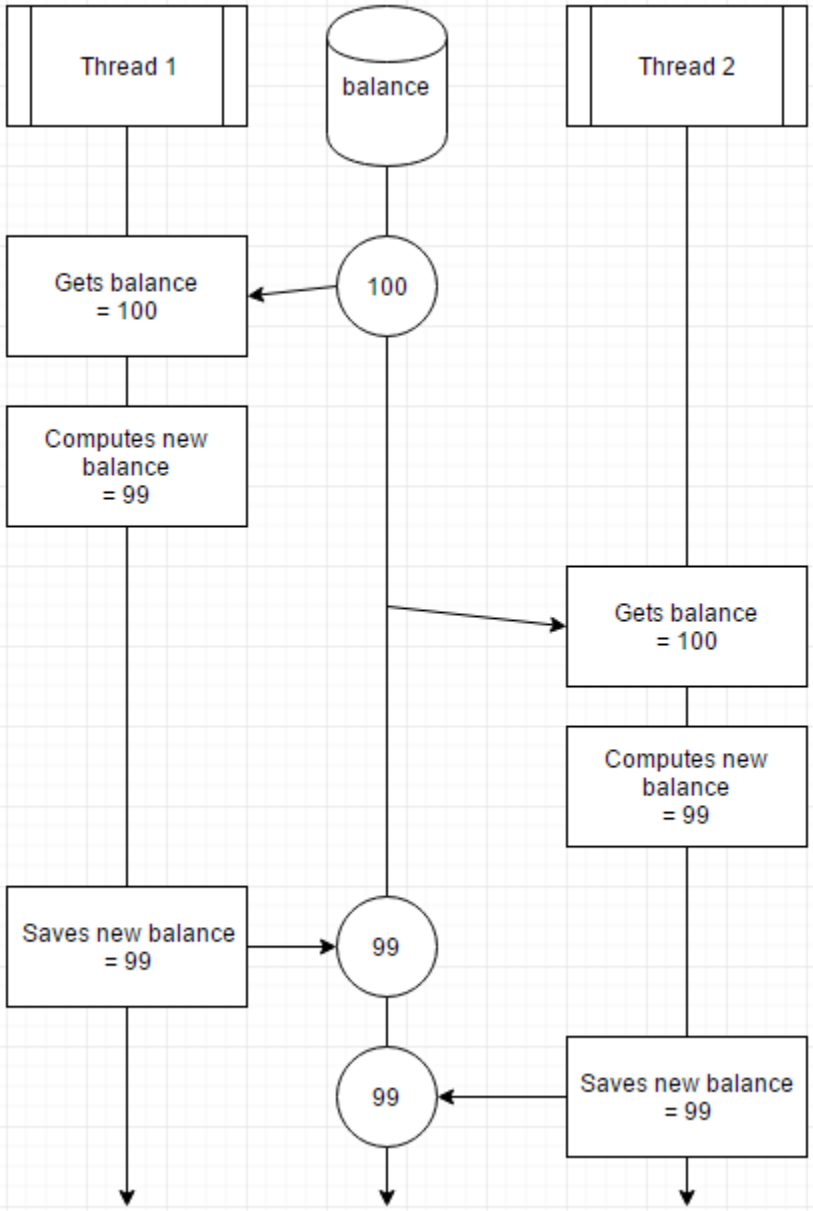
# Recall:

- The scheduler is non-deterministic
  - And completely unpredictable
- It could pre-empt a thread at any time
- Even when it is in the middle of a statement

# Race Condition

- We say that there is a "***race condition***" when one thread must race to finish it's work before another thread tries to access the data it's working on
- Two threads can race against one another and undo each others work

```
12    public void run() {
13      for(int i = 0; i < WITHDRAWL_PER_THREAD; i++) {
14        // balance = balance - 1;
15        balance--;
16      }
17    }
```




- Both threads "withdrew" $1
- So $2 were withdrawn
- Balance should be $98

- But In the end, the balance was $99

```java
public class ListRaceRunnable implements Runnable {

  public final static int NUM_THREADS = 10;
  public final static int TO_ADD_PER_THREAD = 1000;

  private final List<Object> list;
  private final int toAdd;

  public ListRaceRunnable(List<Object> aList, int aToAdd) {
    list = aList;
    toAdd = aToAdd;
  }

  public void run() {
    System.out.println(Thread.currentThread().getName() + " starting");
    for(int i = 0; i < toAdd; i++) {
      list.add(new Object());
    }
    System.out.println(Thread.currentThread().getName() + " ending");
  }

  public static void main(String[] args) throws Exception {
    System.out.println();
    List<Object> list = new ArrayList<Object>();
    Thread[] ts = new Thread[NUM_THREADS];
    for(int i = 0; i < ts.length; i++) {
      ts[i] = new Thread(new ListRaceRunnable(list, TO_ADD_PER_THREAD), "" + i);
      ts[i].start();
    }
    for(Thread t : ts) {
      t.join();
    }
    System.out.println("Expecting: "
      + NUM_THREADS*TO_ADD_PER_THREAD + " elements");
    System.out.println("   Actual: " + list.size() + " elements");
    System.out.println();
  }
}
```

```
0 starting
2 starting
1 starting
3 starting
Exception in thread "2" 4 starting
Exception in thread "1" 3 ending
5 starting
4 ending
Exception in thread "0" 5 ending
java.lang.ArrayIndexOutOfBoundsException: 1851
6 starting      at java.util.ArrayList.add(ArrayList.java:459)

8 starting
        at ListRaceRunnable.run(ListRaceRunnable.java:20)9 starting

7 starting
6 ending        at java.lang.Thread.run(Thread.java:745)

8 ending
java.lang.ArrayIndexOutOfBoundsException: 497 ending

        at java.util.ArrayList.add(ArrayList.java:459)9 ending

        at ListRaceRunnable.run(ListRaceRunnable.java:20)
        at java.lang.Thread.run(Thread.java:745)
java.lang.ArrayIndexOutOfBoundsException: 2045
        at java.util.ArrayList.add(ArrayList.java:459)
        at ListRaceRunnable.run(ListRaceRunnable.java:20)
        at java.lang.Thread.run(Thread.java:745)
Expecting: 10000 elements
   Actual: 8015 elements
```

# ArrayList

```
451     /**
452      * Appends the specified element to the end of this list.
453      *
454      * @param e element to be appended to this list
455      * @return <tt>true</tt> (as specified by {@link Collection#add})
456      */
457     public boolean add(E e) {
458         ensureCapacityInternal(size + 1);  // Increments modCount!!
459         elementData[size++] = e;
460         return true;
461     }
```

# Approaches

1. Thread synchronization (most **common**)
   - a/k/a mutual exclusion
   - Idea: use a synchronization signal to protect “critical sections” from concurrent modification
   - Locks, Synchronize, Conditions, Monitors, Semaphores, etc.
2. Active objects (entering mainstream: Android)
   - An object has its own thread
   - Only that thread may modify its state
3. Strict object ownership (lesser variant of active object)
4. Atomic operations (seen as **exotic**)
   - Rely on a hardware instruction called “test and set”
   - Make some batch assignment an indivisible unit
     - It either happens in full, or not at all
5. Also: thread-safe wrappers
6. Or: redesign your data structures
   - Best solution if possible

# Synchronized Wrapper

```
12 >  public ListRaceRunnable_Wrap(List<Object> aList, int aToAdd) {…
16
17    public void run() {
18      System.out.println(Thread.currentThread().getName() + " starting");
19      for(int i = 0; i < toAdd; i++) {
20        list.add(new Object());
21      }
22      System.out.println(Thread.currentThread().getName() + " ending");
23    }
24
25    public static void main(String[] args) throws Exception {
26      System.out.println();
27      List<Object> list = Collections.synchronizedList(new ArrayList<Object>());
28      Thread[] ts = new Thread[NUM_THREADS];
29      for(int i = 0; i < ts.length; i++) {
30        ts[i] = new Thread(new ListRaceRunnable_Wrap(list, TO_ADD_PER_THREAD), "" + i);
31        ts[i].start();
32      }
```

# Synchronized Keyword

```
12 ›   public ListRaceRunnable_Synch(List<Object> aList, int aToAdd) {⋯
16
17 ⌄   public void run() {
18       System.out.println(Thread.currentThread().getName() + " starting");
19 ⌄     for(int i = 0; i < toAdd; i++) {
20 ⌄       synchronized (list) {
21           list.add(new Object());
22         }
23       }
24       System.out.println(Thread.currentThread().getName() + " ending");
25     }
26
27 ⌄   public static void main(String[] args) throws Exception {
28       System.out.println();
29       List<Object> list = new ArrayList<Object>();
30       Thread[] ts = new Thread[NUM_THREADS];
31 ⌄     for(int i = 0; i < ts.length; i++) {
32         ts[i] = new Thread(new ListRaceRunnable_Synch(list, TO_ADD_PER_THREAD), "" + i);
33         ts[i].start();
34       }
```

# Lock

```
10    private final static Lock LOCK = new ReentrantLock();
```

```
15    public ListRaceRunnable_Lock(List<Object> aList, int aToAdd) {…
19
20    public void run() {
21      System.out.println(Thread.currentThread().getName() + " starting");
22      for(int i = 0; i < toAdd; i++) {
23        LOCK.lock();
24        try {
25          list.add(new Object());
26        } finally {
27          LOCK.unlock();
28        }
29      }
30      System.out.println(Thread.currentThread().getName() + " ending");
31    }
32
33    public static void main(String[] args) throws Exception {
34      System.out.println();
35      List<Object> list = new ArrayList<Object>();
36      Thread[] ts = new Thread[NUM_THREADS];
37      for(int i = 0; i < ts.length; i++) {
38        ts[i] = new Thread(new ListRaceRunnable_Lock(list, TO_ADD_PER_THREAD), "" + i);
39        ts[i].start();
40      }
```

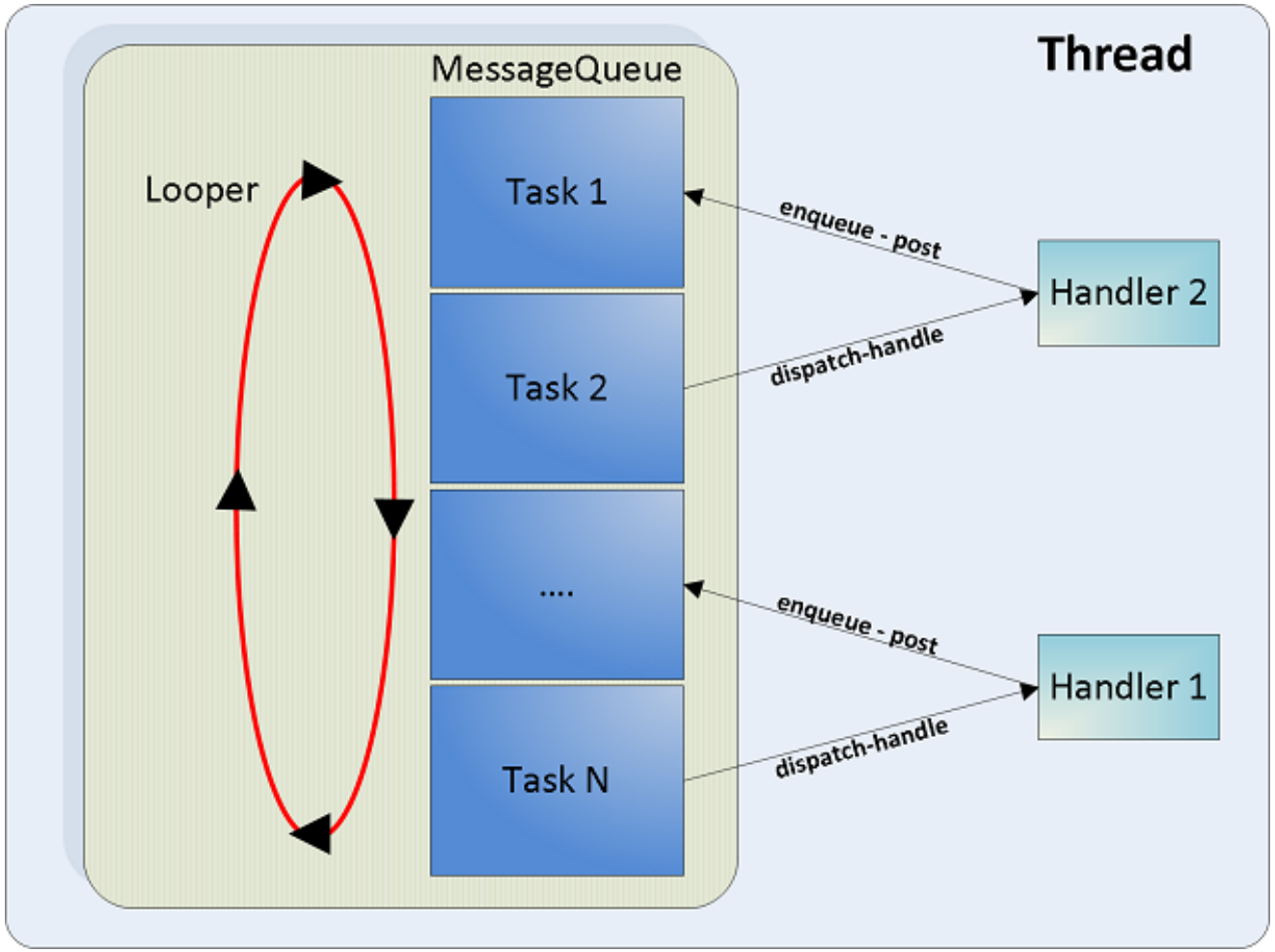
# Atomic

```
10    private final AtomicReference<List<Object>> list;
11    private final int toAdd;
12
13 ›  public ListRaceRunnable_Atomic(⋯
18
19    public void run() {
20      System.out.println(Thread.currentThread().getName() + " starting");
21      for(int i = 0; i < toAdd; i++) {
22        List<Object> oldList;
23        List<Object> newList;
24        do {
25          oldList = list.get();
26          newList = new ArrayList<Object>(oldList);
27          newList.add(new Object());
28        } while(!list.compareAndSet(oldList, newList));
29      }
30      System.out.println(Thread.currentThread().getName() + " ending");
31    }
32
33    public static void main(String[] args) throws Exception {
34      System.out.println();
35      AtomicReference<List<Object>> list =
36        new AtomicReference<List<Object>>(new ArrayList<Object>());
37      Thread[] ts = new Thread[NUM_THREADS];
38      for(int i = 0; i < ts.length; i++) {
39        ts[i] = new Thread(new ListRaceRunnable_Atomic(list, TO_ADD_PER_THREAD), "" + i);
40        ts[i].start();
41      }
```

# Atomic

```
4  public class BankRaceRunnable_atomic implements Runnable {
5
6    private final static int NUM_THREADS = 2;
7    private final static int INITIAL_BALANCE = 10000;
8    private final static int WITHDRAWL_PER_THREAD = INITIAL_BALANCE / NUM_THREADS;
9    private final static int EXPECTED_WITHDRAWN = NUM_THREADS * WITHDRAWL_PER_THREAD;
10
11   private static AtomicInteger balance = new AtomicInteger(INITIAL_BALANCE);
12
13   public void run() {
14     for(int i = 0; i < WITHDRAWL_PER_THREAD; i++) {
15       // balance = balance - 1;
16       // balance.decrementAndGet()
17       balance.addAndGet(-1);
18     }
19   }
20
21   public static void main(String[] args) throws InterruptedException {
22     Thread[] ts = new Thread[NUM_THREADS];
23     for(int i = 0; i < NUM_THREADS; i++) {
24       ts[i] = new Thread(new BankRaceRunnable_atomic());
25       ts[i].start();
26     }
```

# Active Object



# Actor Model

- The **actor model** in computer science is a mathematical model of concurrent computation that treats "actors" as the universal primitives of concurrent computation.
- In response to a message that it receives, an actor can: make local decisions, create more actors, send more messages, and determine how to respond to the next message received.
- Actors may modify private state, but can only affect each other through messages (avoiding the need for any locks). * Wikipedia

```
class OriginalClass {

    private double val = 0.0;

    void doSomething() {
        val = 1.0;
    }

    void doSomethingElse() {
        val = 2.0;
    }
}
```

* https://en.wikipedia.org/wiki/Active_object

# Java 5-7

```
class BecomeActiveObject {

    private double val = 0.0;

    private BlockingQueue<Runnable> dispatchQueue = new LinkedBlockingQueue<Runnable>();

    public BecomeActiveObject() {
        new Thread (new Runnable() {

                @Override
                public void run() {
                    while(true) {
                        try {
                            dispatchQueue.take().run();
                        } catch (InterruptedException e) {
                            // okay, just terminate the dispatcher
                        }
                    }
                }
            }
        ).start();
    }
```

# Java 5-7

```
//
void doSomething() throws InterruptedException {
    dispatchQueue.put(new Runnable() {
            @Override
            public void run() {
                val = 1.0;
            }
        }
    );
}

//
void doSomethingElse() throws InterruptedException {
    dispatchQueue.put(new Runnable() {
            @Override
            public void run() {
                val = 2.0;
            }
        }
    );
}
```

# Note*

- This is how GUI updates happen

- When you say “repaint” you are sending a message to the UI thread telling it that it should repaint the UI for the component on which repaint was called.

# Java 8

```
public class AnotherActiveObject {
    private double val;

    // container for tasks
    // decides which request to execute next
    // asyncMode=true means our worker thread processes its local task queue in the FIFO order
    // only single thread may modify internal state
    private final ForkJoinPool fj = new ForkJoinPool(1, ForkJoinPool.defaultForkJoinWorkerThreadFactory, null, true);

    // implementation of active object method
    public void doSomething() throws InterruptedException {
        fj.execute(() -> {val = 1.0;});
    }

    // implementation of active object method
    public void doSomethingElse() throws InterruptedException {
        fj.execute(() -> {val = 2.0;});
    }
}
```

# For you

- Focus on **Locks**!
- Only use the others if you have a good reason and you are sure it's the right thing to do
  - Atomic for primitives
- Unless you're in Android, then use active objects

# Nomenclature

- When an object can be used safely by multiple threads, we say that it is:
  - ***Thread-safe***

- Most things are not thread-safe by default
  - There is significant overhead to make them thread-safe
  - Locks are not free
  - Atomic actions are not free
  - Message passing (active object) is not free

# Recap

- Race Conditions
  - Scheduling
  - Bank
  - List

- Approaches
  - **Locks**
  - Wrappers
  - Synchronized
  - Active Objects
    - Actor model
  - Atomic operations
  - Ownership
  - Redesign

- Next: Deadlock