Object Oriented Programming and Design in Java

Session 17
Instructor: Bert Huang
Announcements

• Homework 3 due now.
• Homework 4 released today.
  Due Mon. Apr. 19
• Final Exam Monday May 10 at 9 AM
Review

• Horstmann’s graph editor framework
• Prototype pattern
• Simple Graph Editor:
  • extended Graph, AbstractEdge
  • implemented Node
Today’s Plan

- Multithreading and Concurrency
- Multithreading in Java
- Handling race conditions
- Handling race conditions in Java
Multithreading

- Modern computer programs perform various calculations simultaneously
- Each parallel program unit is called a thread
- In most cases, threads are not actually run in parallel, but by taking turns
- But the OS is responsible for the turn-taking; we don’t know its policy
Processes vs. Threads

- Modern OS distinguish processes from threads
- Threads share memory
- Processes don’t share memory

Firefox
- Window 1
- Window 2
- Downloads

Eclipse IDE
- text editor
- javadoc viewer
Threads in Java

- java.lang.Thread

- **Construct with** `Thread(Runnable target)`

- **interface** `Runnable` **has a single method:**
  ```java
  void run()
  ```

- **Thread:** `start()`, `sleep(long millis)`,
  `interrupt()`, `yield()`, `join()`
public class GreetingProducer implements Runnable {
    public GreetingProducer(String aGreeting) {
        greeting = aGreeting;
    }

    public void run() {
        try {
            for (int i = 1; i <= REPETITIONS; i++) {
                System.out.println(i + " : " + greeting);
                Thread.sleep(DELAY);
            }
        } catch (InterruptedException exception) { }
    }

    private String greeting;
    private static final int REPETITIONS = 10;
    private static final int DELAY = 100;
}
This program runs two threads in parallel.

```java
public class ThreadTester {
    public static void main(String[] args) {
        Runnable r1 = new GreetingProducer("Hello, World!");
        Runnable r2 = new GreetingProducer("Goodbye, World!");

        Thread t1 = new Thread(r1);
        Thread t2 = new Thread(r2);

        t1.start();
        t2.start();
    }
}
```
Interrupting Threads

- If you need to terminate a thread, call Thread.interrupt()
- Causes Thread.sleep() to throw InterruptedException
- Your run method should be structured to handle interrupts cleanly

```java
public void run() {
    try {
        while (more_work_to_do) {
            // do work
            Thread.sleep(DELAY);
        }
    } catch (InterruptedException e) {
        // clean up
    }
}
```
Joining Threads

- myThread.join() joins Thread myThread with the current thread
- i.e., waits for myThread to finish its run() method
Race Conditions

- Multiple threads can modify the same memory
- Race condition: when poor timing causes threads to modify memory with unexpected results
- Usually involving multiple threads “racing” to modify the memory first
Incrementing a Counter

- Thread 0: \( c = c + 1; \)
- Thread 1: \( c = c + 1; \)
- The operation reads current value of \( c \)
- Sets \( c \) to that value + 1

<table>
<thead>
<tr>
<th></th>
<th>( c ): 0</th>
<th>( c ): 1</th>
<th>( c ): 2</th>
<th>( c ): 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>T0:</td>
<td>( c ) is 0</td>
<td>set ( c ) to 1</td>
<td>( c ) is 1</td>
<td>set ( c ) to 2</td>
</tr>
<tr>
<td>T1:</td>
<td>( c ) is 2</td>
<td>set ( c ) to 3</td>
<td>( c ) is 2</td>
<td>set ( c ) to 3</td>
</tr>
</tbody>
</table>
Incrementing a Counter

- Both threads can read at the same time, and set \( c \) to \( c + 1 \)
- Result should be \( c + 2 \), but instead is \( c + 1 \)
Locks

- We can use *locks* to fix race conditions
- Threads temporarily acquire ownership of locks
- Only one thread can own a lock at a time
- If a thread tries to acquire a lock but it is owned by another, it waits
- When a lock owner releases the lock, all waiting threads are notified
Lock Interface

- `java.util.concurrent.locks` package includes the `Lock` interface

- Objects that implement `Lock` have
  - `lock()` // prevent other threads from locking this object
  - `unlock()` // allow other threads to lock this object
import java.util.ArrayList;

/**
 * Running multiple threads of this on the same list will cause
 * race conditions
 */
public class UnsafeAdder implements Runnable {
    public UnsafeAdder(ArrayList<Integer> a) {
        list = a;
    }

    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                list.add(i);
                Thread.sleep(10);
            }
        } catch (InterruptedException e) {} 
    }

    private ArrayList<Integer> list;
}
public class SafeAdder implements Runnable {
    public SafeAdder(ArrayList<Integer> a, Lock myLock) {
        list = a;
        lock = myLock;
    }

    public void run() {
        try {
            for (int i = 0; i < 10; i++) {
                lock.lock();
                try {
                    list.add(i);
                } finally {
                    lock.unlock(); // Guaranteed to unlock even if
                                    // list.add(i) throws an exception
                }
            }
            Thread.sleep(10);
        } catch (InterruptedException e) {} 
    }

    private Lock lock;
    private ArrayList<Integer> list;
}
public class LockTest {
    public static void main(String[] args) {
        ArrayList<Integer> a = new ArrayList<Integer>();
        Thread t1 = new Thread(new UnsafeAdder(a));
        Thread t2 = new Thread(new UnsafeAdder(a));
        t1.start(); t2.start();
        try {
            t1.join(); t2.join();
        } catch (InterruptedException e) {} 
        System.out.println("No lock: "+ a);

        Lock lock = new ReentrantLock();
        ArrayList<Integer> b = new ArrayList<Integer>();
        Thread t3 = new Thread(new SafeAdder(b, lock));
        Thread t4 = new Thread(new SafeAdder(b, lock));
        t3.start(); t4.start();
        try {
            t3.join(); t4.join();
        } catch (InterruptedException e) {} 
        System.out.println("With lock: "+ b);
    }
}

No lock: [0, 0, 1, 2, 3, 4, 5, 6, 7, 8, null, 9]
With lock: [0, 0, 1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6, 7, 7, 8, 8, 9, 9]
Producer/Consumer

- Common pattern in threaded programs
- Some threads produce resources, other consume resources
  - e.g., producers add elements to a set while consumers remove elements
- Consumers must wait until set is nonempty
  - Locks are not enough to make this work
Consumer Attempt 1

- while set is empty
  
  ```java
  Thread.sleep(DELAY)
  setLock.lock()
  consume(set.remove())
  setLock.unlock()
  ```

- JVM could switch to another thread after passing through while check

- Then when this thread resumes, set could be empty
Consumer Attempt 2

- setLock.lock()
  while (set is empty)
    Thread.sleep(DELAY)
  consume(set.remove())
  setLock.unlock()

- While this thread is waiting for the set to be non-empty, no one else can lock()
Condition Objects

- Each `Lock` can have any number of `Condition` objects
- `Condition setNonEmpty = setLock.newCondition()`
- `setLock.lock()`
  - `while(set.isEmpty())`
  - `setNonEmpty.await() // releases the lock`
- Whenever the condition could have changed, call `setNonEmpty.signalAll()`
- Unblock all waiting threads, but a thread must reacquire the lock before returning from `await`
Object Locks

- Java Objects have built-in locks
- Any method tagged with keyword synchronized requires a lock
- When the method finishes, the lock is automatically released
- Object locks also allow the command wait(), used to wait for a condition
- After a condition changes, call notifyAll()
Object Locked

ArrayList<E>

- public synchronized E remove()
  {
    while (size == 0) wait();
    ...
  }

- public synchronized void add(E obj)
  {
    ...
    notifyAll();
  }
Dining Philosophers
Threads

- Multithreading allows our programs to perform tasks in parallel
- But requires coordination of the threads' memory operations
- Coordinate threads using locks and conditions
  - Lock interface
  - Object locks (synchronized methods)
Reading

- Horstmann 9.1-9.2