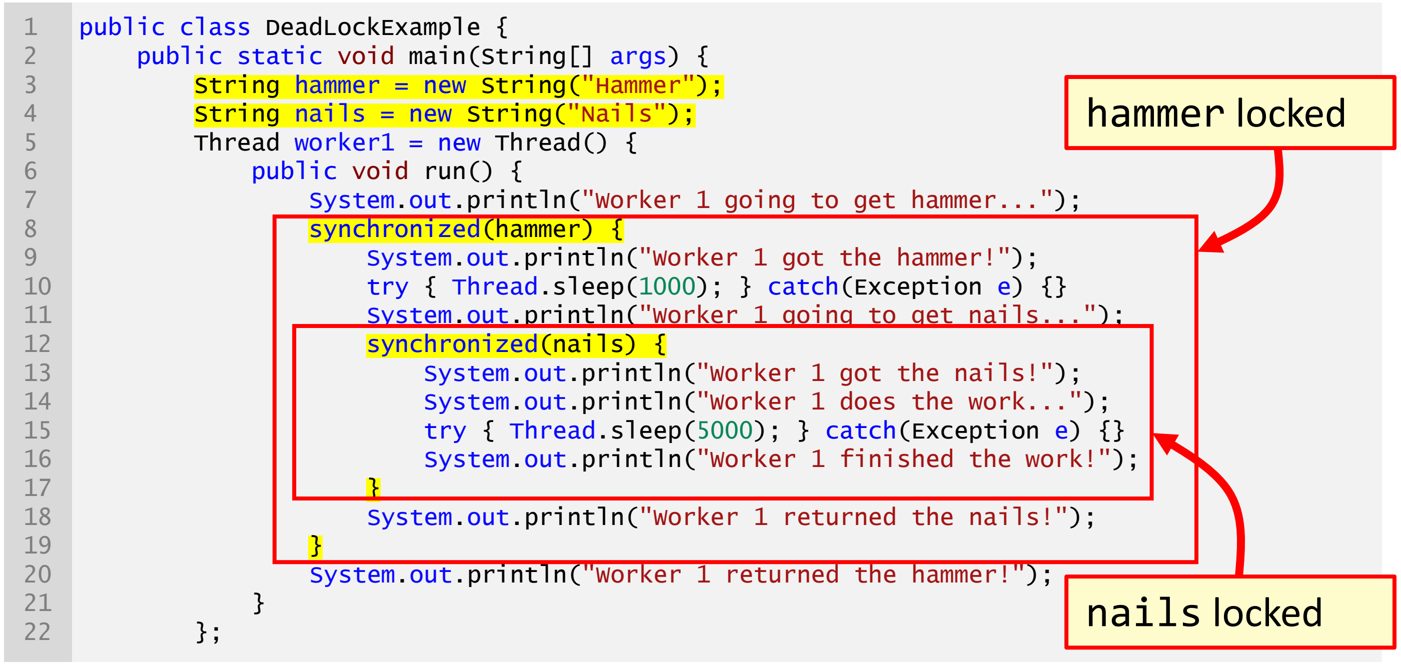
JC2002 Java Programming

Lecture 30: Liveness and high-level concurrency

Liveness

- The ability of an application to execute in a timely manner is known as its *liveness*
- Liveness can be compromised by *deadlocks*, *starvation* and *livelocks*
  - *Deadlocks* happen when two threads are blocking each other
  - *Starvation* happens when low priority thread cannot access shared resources, because they are reserved by high priority “greedy” threads
  - *Livelock* is similar to deadlock, but the threads are not blocked indefinitely, they are just too slow to respond to each other

Deadlock example: worker 1



```java
public class DeadLockExample {
  public static void main(String[] args) {
    String hammer = new String("Hammer");
    String nails = new String("Nails");
    Thread worker1 = new Thread() {
      public void run() {
        System.out.println("Worker 1 going to get hammer...");
        synchronized(hammer) {
          System.out.println("Worker 1 got the hammer!");
          try { Thread.sleep(1000); } catch(Exception e) {}
          System.out.println("Worker 1 going to get nails...");
          synchronized(nails) {
            System.out.println("Worker 1 got the nails!");
            System.out.println("Worker 1 does the work...");
            try { Thread.sleep(5000); } catch(Exception e) {}
            System.out.println("Worker 1 finished the work!");
          }
          System.out.println("Worker 1 returned the nails!");
        }
        System.out.println("Worker 1 returned the hammer!");
      }
    };
```

Deadlock example: worker 2

```
    Thread worker2 = new Thread() {
      public void run() {
        System.out.println("Worker 2 going to get nails...");
        synchronized(nails) {
          System.out.println("Worker 2 got the nails!");
          try { Thread.sleep(500); } catch(Exception e) {}
          System.out.println("Worker 2 going to get hammer...");
          synchronized(hammer) {
            System.out.println("Worker 2 got the nails!");
            System.out.println("Worker 2 does the work...");
            try { Thread.sleep(5000); } catch(Exception e) {}
            System.out.println("Worker 2 finished the work!");
          }
          System.out.println("Worker 2 returned the hammer!");
        }
        System.out.println("Worker 2 returned the nails!");
      }
    };
    worker1.start();
    worker2.start();
  }
}
```

nails locked

hammer locked

Deadlock example: output

```
$ java DeadLockExample
Worker 1 going to get hammer...
Worker 1 got the hammer!
Worker 2 going to get nails...
Worker 2 got the nails!
Worker 2 going to get hammer...
Worker 1 going to get nails...
```

- The program is deadlocked, because worker 1 has the hammer and worker 2 has the nails, and neither worker can proceed…

Avoiding deadlocks

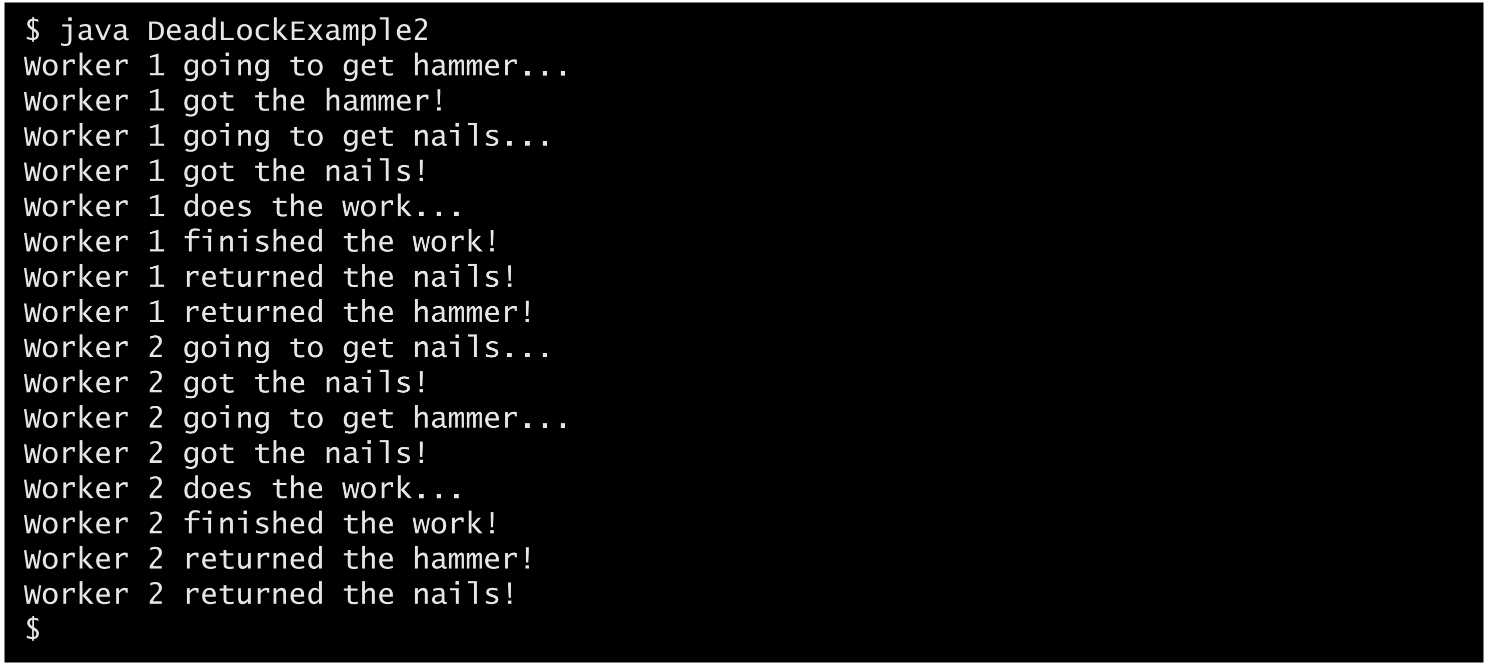
- Avoid nested locks (`synchronization` blocks inside each other)
- Avoid unnecessary locks: only lock objects you really need
- Instead of locking objects via synchronization, use *immutable objects* whenever possible
  - An object is *immutable* if its state cannot be changed after constructed
  - Do not provide setter methods, define all fields final and private
- Invoke **`t.join()`** method of Thread `t` to make other threads to start after `t` has finished
  - Timed version `join(m)` waits at most *m* milliseconds for thread to die

Example of avoiding deadlocks with join()

```
public class DeadLockExample2 {
    public static void main(String[] args) {
        String hammer = new String("Hammer");
        String nails = new String("Nails");
...
...
        try {
            worker1.start();
            worker1.join();
            worker2.start();
        }
        catch(InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Same as previous example

Avoiding deadlocks with join(): output



```
$ java DeadLockExample2
Worker 1 going to get hammer...
Worker 1 got the hammer!
Worker 1 going to get nails...
Worker 1 got the nails!
Worker 1 does the work...
Worker 1 finished the work!
Worker 1 returned the nails!
Worker 1 returned the hammer!
Worker 2 going to get nails...
Worker 2 got the nails!
Worker 2 going to get hammer...
Worker 2 got the nails!
Worker 2 does the work...
Worker 2 finished the work!
Worker 2 returned the hammer!
Worker 2 returned the nails!
$
```

High level concurrency

- Concurrency explained so far on this course is based on low-level API useful for basic tasks, but not suitable for more advanced tasks
- Package `java.util.concurrent` offers more advanced features:
    - *Lock* objects for more sophisticated synchronization features
    - *Executors* defining high level API for launching and managing threads
    - *Concurrent collections* for managing and synchronizing large collections of data
    - *Atomic variables* for atomic operations without synchronization

Lock objects

- The main advantage of the lock objects is their ability to back out of an attempt to acquire a lock
- Method **`tryLock()`** can be used to try to lock a lock object, it returns false if locking is not possible (someone acquired lock already)
- It is also possible to use timed version of **`tryLock(m)`**, that waits for the given timeout `m` (in milliseconds) before giving up
- And other advanced features (out of the scope of this course)

Avoiding deadlocks with Lock objects

```java
import java.util.concurrent.locks.ReentrantLock;
public class LockExample {
    public static void main(String[] args) {
        ReentrantLock hammerLock = new ReentrantLock();
        ReentrantLock nailLock = new ReentrantLock();
        Thread worker1 = new Thread() {
            public void run() {
                System.out.println("Worker 1 going to get hammer...");
                if(!hammerLock.tryLock()) {
                    System.out.println("Hammer already taken!");
                    return;
                }
                System.out.println("Worker 1 got the hammer!");
                try { Thread.sleep(1000); } catch(Exception e) {}
...             // ... Try locking nails in the same way
                System.out.println("Worker 1 got the nails!");
                System.out.println("Worker 1 does the work...");
                try { Thread.sleep(5000); } catch(Exception e) {}
                System.out.println("Worker 1 finished the work!");
                nailLock.unlock();
                System.out.println("Worker 1 returned the nails!");
...
```

Thread `worker2` implemented in a similar fashion

Lock example: output

```
$ java LockExample
Worker 1 going to get hammer...
Worker 2 going to get nails...
Worker 1 got the hammer!
Worker 1 going to get nails...
Worker 2 got the hammer!
Worker 2 going to get hammer...
Nails already taken!
Hammer already taken!
$
```

- Items already taken are detected and deadlock avoided!

Executor interfaces

- The executor interface in `java.util.concurrent` package provides methods for launching and managing tasks (e.g., threads)
- Assuming `r` is a `Runnable` and `e` is an **`Executor`** object, it is possible to replace `(new Thread(r)).start();` with `e.execute(r);`
- Most of the executor implementations are designed to handle *thread pools* that consist of several *worker threads*
  - Advantage in large scale applications, such as web servers that need to coordinate a large number of threads in a scalable manner

Simple executor example (1)

```java
import java.util.concurrent.*;
import java.util.*;
class MyThread implements Runnable {
    int threadNum, start, end;
    MyThread(int num, int start, int end) {
        this.threadNum = num; this.start = start; this.end = end;
    }
    public void run() {
        try {
            for(int i = start; i <= end; i++) {
                System.out.printf("Thread #%d, step %d\n", threadNum,i);
                Random rand = new Random();
                Thread.sleep(rand.nextInt(1000));
            }
        }
        catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
}
```

Implement custom thread in a normal way

UNIVERSITY OF ABERDEEN

Simple executor example (2)

```java
public class ExecutorExample {
    public static void main(String[] args) {
        ExecutorService executor = Executors.newFixedThreadPool(10);
        Random rand = new Random();
        for(int i=0; i<5; i++) {
            int start = rand.nextInt(100);
            int end = start + rand.nextInt(3) + 1;
            MyThread thread = new MyThread(i+1,start,end);
            executor.execute(thread);
        }
        executor.shutdown();
    }
}
```

Create five threads with different random characteristics, and execute them via the executor object

Simple executor example: output

```
$ java ExecutorExample
Thread #1, step 46
Thread #5, step 19
Thread #4, step 49
Thread #3, step 49
Thread #2, step 24
Thread #3, step 50
Thread #2, step 25
Thread #1, step 47
Thread #2, step 26
Thread #3, step 51
Thread #3, step 52
Thread #4, step 50
Thread #5, step 20
Thread #4, step 51
$
```

Final remarks on advanced concurrency

- Concurrency is a very complex topic, especially when multicore platforms are concerned
- For most programmers, the low-level API is sufficient, but for more advanced applications dealing with a lot of of data and threads, the high-level API from `java.util.concurrent` is a necessity
- For more details:
  - https://docs.oracle.com/javase/tutorial/essential/concurrency/procthread.html
  - **Book:** Brian Goetz et al.: *Java Concurrency in Practice* (Addison-Wesley)

Summary

- In concurrent programming, blocks of code are executed simultaneously, typically by using multiple threads
  - In Java, threads are defined and used by extending class `Thread` or implementing interface Runnable
  - Swing provides also API for using *worker threads* to run heavy tasks on the background
- Multithreading requires programmer to consider problems with thread interference and deadlocks
  - Thread interference can be avoided using locks, but this may lead to deadlocks and other liveness problems
  - Some advice was given to avoid deadlocks

Questions, comments?