SE 552: Lecture 2

Overview

Thread API

Locks

Wait/notify next week!

A comment on coding style

Some rules I've used for coding:

- No public classes, only public interfaces! (Except where Java forces them.)
- Declare all variables and fields `final` unless absolutely necessary.
- Use lots of debugging println statements.
- Javadoc comments for all public fields and methods.

What are the pros and cons of these coding rules?

Thread API

Creating new threads...

```
class HW implements Runnable {
  public void run () {
    System.out.println ("Hello");
    System.out.println ("World");
  }
}
```

What does this code do?

```
HW hello = new HW ();
Thread t1 = new Thread (hello);
t1.start ();
```

Thread API

```
class HW implements Runnable {
  public void run () {
    System.out.println ("Hello");
    System.out.println ("World");
  }
}
```

What about this code?

```
HW hello = new HW ();
Thread t1 = new Thread (hello);
Thread t2 = new Thread (hello);
t1.start ();
t2.start ();
```

Thread API

```
class HWMany implements Runnable {
  public void run () {
    while (true) {
      System.out.println ("Hello");
      System.out.println ("World");
      sleep for a bit
    }
  }
}
```

What about this code?

```
HW hello = new HWMany ();
Thread t1 = new Thread (hello);
Thread t2 = new Thread (hello);
t1.start ();
t2.start ();
```

How do we implement *sleep for a bit*?

Thread API

Methods in the Thread API

- run ()
- start ()
- sleep (milliseconds)
- interrupt ()
- currentthread ()
- join ()

What do these methods do? (Think ‘state machine’!)

What exceptions can these methods throw?

Invariants and safety

```
class BankBalance {
  // INVARIANT: balance = deposits - withdrawls
  protected int balance;
  protected int withdrawls;
  protected int deposits;
  public deposit (int amount) {
    balance = balance + amount;
    deposits = deposits + amount;
  }
  public withdraw (int amount) {
    balance = balance - amount;
    withdrawls = withdrawls + amount;
  }
  public int balance () { return balance; }
  ...
}
```

What is an invariant?

How can we break the invariant for BankBalance?
How can we fix this?

Critical sections and atomicity

```
class BankBalance {
  // INVARIANT: balance = deposits - withdrawls
  protected int balance;
  protected int withdrawls;
  protected int deposits;
  public deposit (int amount) {
    balance = balance + amount;
    deposits = deposits + amount;
  }
  public withdraw (int amount) {
    balance = balance - amount;
    withdrawls = withdrawls + amount;
  }
  public int balance () { return balance; }
  ...
}
```

What is a critical section? What are the critical sections of this code?

What is an atomic operation? How do locks help make critical sections atomic?

Locks in Java

```
Object lock = new Object ();
...
synchronized (lock) {
  critical section
}
```

What happens when a thread enters a synchronized section?

What happens when a thread leaves a synchronized section?

Locks in Java

What happens if thread T executes:

```
synchronized (lock) {
  critical section A
  synchronized (lock) {
    critical section B
  }
  critical section C
}
```

Who owns the lock when T executes A?

Who owns the lock when T executes B?

Who owns the lock when T executes C?

Locks in Java

Shorthand:

```
synchronized void foo () {
  critical section
}
```

is shorthand for:

```
void foo () {
  synchronized (this) {
    critical section
  }
}
```

What are the pros and cons of using this shorthand?

Locks in Java

How are locks implemented?

Locks example

An unsafe class for buffers: UnsafeBuffer.java

A test program for this class: TestBuffer.java

What are the invariants for a buffer?

Can we get this test program to violate the invariants of the buffer class?

This is one thing step debuggers are really good at!

Summary

We've now seen two-thirds of Java's concurrency features.

Next week: wait/notify and immutability.

After that: more exclusion, then the rest of Lea.

Homework: Sheet 1, due before next lecture.

Reading: Chapter 2 of Lea, especially material on immutability.