Concurrency in Java

Brad Vander Zanden

Processes and Threads

- Process: A self-contained execution environment
- Thread: Exists within a process and shares the process's resources with other threads

Java's Thread Mechanism

- Low Level
  - Thread Class
  - Runnable Interface
- High Level: Thread executors and tasks

Runnable Interface

```
public class HelloRunnable implements Runnable {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new Thread(new HelloRunnable())).start();
    }

}
```

Subclassing Thread

```
public class HelloThread extends Thread {

    public void run() {
        System.out.println("Hello from a thread!");
    }

    public static void main(String args[]) {
        (new HelloThread()).start();
    }

}
```

Thread vs. Runnable

- Runnable allows you to subclass another object
- Thread is more direct and a bit simpler

Pacing a Thread

- Thread.sleep(ms) suspends execution for the specified period
  - gives up processor
  - allows thread to pace execution, such as when doing an animation

Handling Interrupts

- Interrupt() method may be invoked on a thread to notify it of an interrupt
- Ways to handle an interrupt
  - Catch *InterruptedException:* Thrown by methods like sleep and wait
  - Call *Thread.interrupted()*
- Interrupt status flag
  - Checked by interrupted
  - Cleared by InterruptedException or by calling interrupted()

Examples

```
for (int i = 0; i < importantInfo.length; i++) {
    // Pause for 4 seconds
    try {
        Thread.sleep(4000);
    } catch (InterruptedException e) {
        // We've been interrupted: no more messages.
        return;
    }
    // Print a message
    System.out.println(importantInfo[i]);
}
```

Examples

```
for (int i = 0; i < inputs.length; i++) {
    heavyCrunch(inputs[i]);
    if (Thread.interrupted()) {
        // We've been interrupted: no more crunching.
        return;
    }
}
```

Join

- The join method allows one thread to wait for the completion of another thread
- Example: t.join() waits for the thread referenced by *t* to finish execution

A Detailed Example

- //docs.oracle.com/javase/tutorial/essential/concurrency/simple.html

Synchronization

- Why we need it
  - Thread interference: contention for shared resources, such as a counter
  - Memory inconsistency: if there is a *happens-before* relationship where thread A relies on thread B performing a write before it does a read
    - joins are a trivial way to handle memory inconsistency

Synchronization Techniques

- Synchronized Methods
- Synchronized Statements/Locks
- Volatile Variables

Synchronized Methods

```
public class SynchronizedCounter {
    private int c = 0;

    public synchronized void increment() {
        c++;
    }

    public synchronized void decrement() {
        c--;
    }

    public synchronized int value() {
        return c;
    }
}
```

Problem w/o Synchronization

- The single expression c++ can be decomposed into three steps:
  1. Retrieve the current value of c.
  2. Increment the retrieved value by 1.
  3. Store the incremented value back in c.

A Bad Interleaving of Operations

- A possible interleaving of Thread A and B
  - Thread A: Retrieve c.
  - Thread B: Retrieve c.
  - Thread A: Increment retrieved value; result is 1.
  - Thread B: Decrement retrieved value; result is -1.
  - Thread A: Store result in c; c is now 1.
  - Thread B: Store result in c; c is now -1.

Synchronized Statements

```
public void addName(String name) {
    synchronized(this) {
        lastName = name;
        nameCount++;
    }
    nameList.add(name);
}
```

Example with Multiple Locks

```
public class MsLunch {
   private long c1 = 0;
   private long c2 = 0;
   private Object lock1 = new Object();
   private Object lock2 = new Object();

   public void inc1() {
      synchronized(lock1) {
         c1++;
      }}

   public void inc2() {
      synchronized(lock2) {
         c2++;
      }}
}
```

Volatile Variables

- Example: volatile int x1;
- Forces any change made by a thread to be forced out to main memory
- Ordinarily threads maintain local copies of variables for efficiency

Synchronized Method vs Volatile Variables

- synchronized methods
  - force *all* of a thread's variables to be updated from main memory on method entry
  - flush all changes to a thread's variables to main memory on method exit
  - obtain and release a lock on the object
- volatile variable
  - only reads/writes one variable to main memory
  - does no locking

Happens-Before Using Wait

- Object.wait(): suspends execution until another thread calls *notifyAll()* or *notify()*
- Must check condition because notifyAll/notify does not specify which condition has changed
  - Use notify for a mutex where only one thread can use the lock
  - Use notifyAll for situations where all threads might be able to usefully continue

Example

Thread 1

```
public synchronized guardedJoy() {
   // keep looping until event we're
   // waiting for happens
   while(!joy) {
      try {
         wait();
      } catch (InterruptedException e)
{}
   }
   System.out.println("Joy and
efficiency have been achieved!");
}
```

Thread 2

```
public synchronized notifyJoy() {
   joy = true;
   notifyAll();
}
```

Producer-Consumer Example

- http://docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html