COMPSCI 230 S2C 2015
Software Design and Construction

Synchronization (cont.)
Lecture 4 of Theme C

Learning Goals for Today

- Develop a stronger understanding of synchronization in Java.
  - Be able to analyse codes with a small number of interactions between a few threads, answering the question "what execution traces are possible?"
- Learn the syntax for synchronized methods
  - What are the disadvantages of this "syntactic sugar"?
- Learn an important design pattern: using a final instance of a collection to synchronize its methods.
  - A simple example: a thread-safe cache

Goetz's "Simple Synchronization Example"

- "Using synchronized blocks allows you
  - to perform a group of related updates as a set
  - without worrying about other threads
    - interrupting or seeing the intermediate results of a computation."
- Do you understand why you should be concerned if
  - Other threads can interrupt a worker thread, or if
  - Other threads can see a worker's intermediate results?

Atomicity, in detail

- Atomic reading:
  - The variables read by a worker (in an atomic task) must be "locked" against changes by other threads -- until the worker has completed the task.
- Atomic writing:
  - A worker's writes must be invisible to other threads until the worker has finished their atomic task – and they must be visible to the next worker who enters this task.
- Atomic completion:
  - While a worker is performing an atomic task, it should not be interrupted by other workers.
  - This is not an absolute prohibition.
  - If a thread is interrupted, it has to start over from the *beginning* of the task – and this slows progress.
  - In an extreme case (called livelock), every worker attempting the task is interrupted by another worker, so the task is never completed!

The top-level structure of Goetz's example

```
public class SyncExample {
  private static Object lockObject = new Object();

  private static class Thread1 extends Thread {
     … // on next slide
  }
  private static class Thread2 extends Thread {
     … // on next slide
  }

  public static void main(String[] args) {
    new Thread1().start();
    new Thread2().start();
  }
}
```

Thread classes in Goetz's example

```
private static class Thread1 extends Thread {
  public void run() {
    synchronized( lockObject ) {
      x = y = 0;
      System.out.println(x);
} } }

private static class Thread2 extends Thread {
  public void run() {
    synchronized( lockObject ) {
      x = y = 1;
      System.out.println(y);
} } }
```

Expected output:

0
1

or

1
0

- A thread cannot execute a synchronized block until it acquires the "lock" on the block's monitor.
- The lock is released when the thread exits the block.
- A lock has at most one owner at any time.
- A thread can own many locks.

Unsynchronized threads

```
private static class Thread1 extends Thread {
  public void run() {
    y = 0;
    x = y = 0;
    x = 0;
    System.out.println(x);
  }
}
private static class Thread2 extends Thread {
  public void run() {
    y = 1;
    x = y = 1;
    x = 1;
    System.out.println(y);
  }
}
```

Expected output:

0
1

or

1
0

or

1
1

or

0
0

Synchronized Methods

- We can synchronize the body of a method using `this` as its lock:

```
public class Point {
  public void setXY( int x, int y ) {
    synchronized (this) {
      this.x = x; this.y = y;
} } }
```

- This is a very common structure, so Java includes the "synchronized method" as syntactic sugar.
  - See http://en.wikipedia.org/wiki/Syntactic_sugar
- The following is equivalent, and is sweeter to read and write.

```
public class Point {
  public synchronized void setXY( int x, int y ) {
    this.x = x; this.y = y;
} }
```

- Warning: sugar is very unhealthy if you don't "eat your vegetables" too!

Dangerous Sugar...

```
public class SyncExample {
  public static class Thingie {
    private Date lastAccess;
    public synchronized void setLastAccess( Date date ) {
      this.lastAccess = date;
  } }
  public static class MyThread extends Thread {
    private Thingie thingie;
    public MyThread( Thingie thingie ) {
      this.thingie = thingie;
    }
    public void run() {
      thingie.setLastAccess( new Date() );
  } }
  public static void main() {
    Thingie thingie1 = new Thingie(),
            thingie2 = new Thingie();
    new MyThread(thingie1).start();
    new MyThread(thingie2).start();
} }
```

- `setLastAccess()` is a synchronized method, so each instance has its own lock.
- This method is unsafe, because the first worker thread can acquire the lock on `thingie1` **at the same time** the second worker acquires a lock on `thingie2`.

Goetz's Advice on Synchronization

- "Because synchronization prevents multiple threads from executing a block at once,
  - it has performance implications, even on uniprocessor systems.
- "It is a good practice to
  - use synchronization around the smallest possible block of code that needs to be protected.
- "Access to local (stack-based) variables never need to be protected,
  - because they are only accessible from the owning thread."

- In other words:
  - if you're concerned about performance, don't use this syntactic sugar (unless the whole method really needs to be "sweet" ;-).

Most Java Classes are not Synchronized!

- Java has nice support for threads, but you have to be very careful whenever multiple threads can access the same object.
- Goetz: "Because synchronization carries a small performance penalty,
  - most general-purpose classes, like the `Collection` classes in `java.util`, do not use synchronization internally.
  - This means that classes like `HashMap` cannot be used from multiple threads without additional synchronization.
- "You can use the `Collections` classes in a multithreaded application
  - by using synchronization every time you access a method in a shared collection.
  - For any given collection, you must synchronize on the same lock each time.
  - A common choice of lock would be the collection object itself.
- "If the documentation for a class does not say that it is thread-safe, then you must assume that it is not."

A Simple Thread-Safe Cache

```
public class SimpleCache {
  private final Map cache = new HashMap();
  public Object load(String objectName) {
    // load the object somehow
  }
  public void clearCache() {
    synchronized( cache ) {
      cache.clear();
  } }
  public Object getObject( String objectName ) {
    synchronized( cache ) {
      Object o = cache.get( objectName );
      if( o == null ) {
        o = load( objectName );
        cache.put( objectName, o );
    } }
    return o;
  }
}
```

- This code is synchronized on a single (final) instance of a cache object.
- The cache.clear() method will never run concurrently with a cache.get() or cache.put().
- Cache updates are atomic: the cache.get()… cache.put() sequence won't be interrupted.

Sharing access to data summary (Goetz)

- "Because the timing of thread execution is nondeterministic, we need to be careful to control a thread's access to shared data.
  - Otherwise, multiple concurrent threads could step on each other's changes and result in corrupted data, or
  - changes to shared data might not be made visible to other threads on a timely basis.
- "By using synchronization to protect access to shared variables,
  - we can ensure that threads interact with program variables in predictable ways.
- "Every Java object can act as a lock, and synchronized blocks can ensure that
  - only one thread executes synchronized code protected by a given lock at one time."

Learning Goals for Today

- Develop a stronger understanding of synchronization in Java.
  - Be able to analyse codes with a small number of interactions between a few threads, answering the question "what execution traces are possible?"
- Learn the syntax for synchronized methods
  - What are the disadvantages of this "syntactic sugar"?
- Learn an important design pattern: using a final instance of a collection to synchronize its methods.
  - A simple example: a thread-safe cache