Overview of Java
Synchronizers (Part 2)

Douglas C. Schmidt
d.schmidt@vanderbilt.edu
www.dre.vanderbilt.edu/~schmidt

Institute for Software
Integrated Systems
Vanderbilt University
Nashville, Tennessee, USA
Learning Objectives in this Part of the Lesson

- Understand the key Java languages features & library classes for synchronizing Java threads in concurrent programs

```
package java.util.concurrent.locks

Added in API level 1

Interfaces and classes providing a framework for locking and waiting for conditions that is distinct from built-in synchronization and monitors. The framework permits much greater flexibility in the use of locks and conditions, at the expense of more awkward syntax. The Lock interface supports locking disciplines that differ in semantics (reentrant, fair, etc), and that can be used in non-block-structured contexts including hand-over-hand and lock reordering algorithms. The main implementation is ReentrantLock.
```

```
package java.util.concurrent

Added in API level 1

Utility classes commonly useful in concurrent programming. This package includes a few small standardized extensible frameworks, as well as some classes that provide useful functionality and are otherwise tedious or difficult to implement. Here are brief descriptions of the main components. See also the java.util.concurrent.locks and java.util.concurrent.atomic packages.
```
Overview of Java Atomic Operations & Variables
Overview of Java Atomic Operations & Variables

- Atomic actions ensure that changes to a variable are always consistent & visible to other threads

See [en.wikipedia.org/wiki/Linearizability](en.wikipedia.org/wiki/Linearizability)
Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions on variables
Overview of Java Atomic Operations & Variables

• Java supports several types of atomic actions on variables
  • Atomic operations
    • Implement “lock-free” synchronization primitives

```java
int compare_and_swap(int *loc, int oldval, int newval) {
    START_ATOMIC();
    int old_loc_val = *loc;
    if (old_loc_val == oldval) {
        *loc = newval;
        END_ATOMIC();
    }
    return old_loc_val;
}

void lock (int *loc) {
    while (compare_and_swap (loc, 0, 1) == 1);
}

void unlock (int *loc) {
    START_ATOMIC();
    *loc = 0;
    END_ATOMIC();
}
```

[See en.wikipedia.org/wiki/Non-blocking_algorithm](en.wikipedia.org/wiki/Non-blocking_algorithm)
Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions on variables
  - Atomic operations
    - Implement “lock-free” synchronization primitives
    - e.g., `compareAndSwapInt()`

Concurrency

And few words about concurrency with `Unsafe.compareAndSwap` methods are atomic and can be used to implement high-performance lock-free data structures.

For example, consider the problem to increment value in the shared object using lot of threads.

First we define simple interface `Counter`:

```java
interface Counter {
    void increment();
    long getCount();
}
```

Then we define worker thread `CounterClient`, that uses `Counter`:

```java
class CounterClient implements Runnable {
    private Counter c;
    private int num;

    public CounterClient(Counter c, int num) {
        this.c = c;
        this.num = num;
    }

    @Override
    public void run() {
        for (int i = 0; i < num; i++) {
            c.increment();
        }
    }
}
```

See mishadoff.com/blog/java-magic-part-4-sun-dot-misc-dot-unsafe
Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions on variables
  - **Atomic operations**
  - **Volatile variables**
    - Ensure a variable is always written to & read from main memory & is not cached

See [en.wikipedia.org/wiki/Volatile_variable#In_Java](en.wikipedia.org/wiki/Volatile_variable#In_Java)
Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions on variables
  - **Atomic operations**
  - **Volatile variables**
  - **Atomic variables**
    - Provide lock-free thread-safe operations on single variables

See docs.oracle.com/javase/tutorial/essential/concurrency/atomicvars.html
Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions on variables
  - Atomic operations
  - Volatile variables
  - Atomic variables
    - Provide lock-free thread-safe operations on single variables
      - e.g., AtomicLong combines aspects of volatile variables with atomic “compare-and-swap” operations

See [docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicLong.html](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/atomic/AtomicLong.html)
Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions on variables
  - Atomic operations
  - Volatile variables
  - Atomic variables
  - LongAdder
    - Allows multiple threads to update a common sum efficiently under high contention

See docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/LongAdder.html
Overview of Java Atomic Operations & Variables

- Java supports several types of atomic actions on variables
  - **Atomic operations**
  - **Volatile variables**
  - **Atomic variables**
  - **LongAdder**

```java
def compare_and_swap(int *loc, int oldval, int newval)
{
    START_ATOMIC();
    int old_loc_val = *loc;
    if (old_loc_val == oldval)
        *loc = newval;
    END_ATOMIC();
    return old_loc_val;
}
```

In Java

The Java programming language also has the `volatile` keyword, but it is used for a somewhat different purpose. When applied to a field, the Java qualifier `volatile` guarantees that:

- In all versions of Java, there is a global ordering on the reads and writes to a volatile variable. This implies that every thread accessing a volatile field will read its current value before continuing, instead of (potentially) using a cached value. (However, there is no guarantee about the relative ordering of volatile reads and writes with regular reads and writes, meaning that it's generally not a useful threading construct.)
- In Java 5 or later, volatile reads and writes establish a happens-before relationship, much like acquiring and releasing a mutex.[9]

Using `volatile` may be faster than a `lock`, but it will not work in some situations.[citation needed] The range of situations in which volatile is effective was expanded in Java 5; in particular, double-checked locking now works correctly.[10]
Overview of Java
Built-in Monitor Objects
Overview of Java Built-in Monitor Objects

- Monitor objects are built into the Java programming language

See [en.wikipedia.org/wiki/Monitor_(synchronization)](en.wikipedia.org/wiki/Monitor_(synchronization))
Overview of Java Built-in Monitor Objects

- Monitor objects are built into the Java programming language
- They are written in C & implemented in the JVM

Overview of Java Built-in Monitor Objects

- They support two primary types of synchronization mechanisms
Overview of Java Built-in Monitor Objects

- They support two primary types of synchronization mechanisms
  - Mutual exclusion

One thread never enters its critical section at the same time that another concurrent thread enters the same critical section

See en.wikipedia.org/wiki/Mutual_exclusion
Overview of Java Built-in Monitor Objects

- They support two primary types of synchronization mechanisms
  - Mutual exclusion
  - Coordination

Ensure interactions between threads occur
- in the right order
- at the right time
- under the right conditions

See docs.oracle.com/javase/tutorial/essential/concurrency/guardmeth.html
Overview of Java Built-in Monitor Objects

• They support two primary types of synchronization mechanisms
  • Mutual exclusion
  • Coordination

Java built-in monitor objects can also be used to implement other types of synchronization
Overview of Java Built-in Monitor Objects

- Any Java object may be used as a monitor object

This may change if/when Value Types are added to Java, as per [cr.openjdk.java.net/~jrose/values/values-0.html](http://cr.openjdk.java.net/~jrose/values/values-0.html)
Overview of Java Built-in Monitor Objects

- Any Java object may be used as a monitor object
- Methods requiring mutual exclusion must be marked as synchronized

Only one thread at a time is allowed to execute within a synchronized “critical section”

See [en.wikipedia.org/wiki/Critical_section](en.wikipedia.org/wiki/Critical_section)
Overview of Java Built-in Monitor Objects

- Any Java object may be used as a monitor object
  - Methods requiring mutual exclusion must be marked as synchronized
- Alternatively, code statements marked as synchronized provide finer locking granularity

```java
void put(String msg) {
    synchronized (this) {
        mList.add(msg);
        ... 
    }
    ...
}
```

See [docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html](docs.oracle.com/javase/tutorial/essential/concurrency/locksync.html)
Overview of Java Built-in Monitor Objects

- Synchronized methods & statements are *not* a complete solution

```java
public void synchronized put(String msg){
    mQ.add(msg);
}

public String synchronized take(){
    return mQ.remove(0);
}
```

Concurrent calls to these methods will "busy wait" or worse..

See [en.wikipedia.org/wiki/Busy_waiting](en.wikipedia.org/wiki/Busy_waiting)
Overview of Java Built-in Monitor Objects

- Synchronized methods & statements are *not* a complete solution
- Java’s built-in monitor objects therefore provide waiting & notification mechanisms

![Diagram of Producer, Consumer, SimpleBlocking Queue, Wait Queue, Entrance Queue with methods and contain markers]
Overview of Java Built-in Monitor Objects

- Synchronized methods & statements are *not* a complete solution
- Java’s built-in monitor objects therefore provide waiting & notification mechanisms

wait() causes the current thread to wait on the monitor condition until another thread invokes the notify() or notifyAll() method for this object.
Synchronized methods & statements are not a complete solution. Java’s built-in monitor objects therefore provide waiting & notification mechanisms.

notify() & notifyAll() wake up a single thread or multiple threads, respectively, that are waiting on this object's monitor condition.
Overview of Java Built-in Monitor Objects

- Synchronized methods & statements are *not* a complete solution
- Java’s built-in monitor objects therefore provide waiting & notification mechanisms

See the upcoming lesson on “Java Built-in Monitor Objects”
End of Overview of Java Synchronizers (Part 2)
1. Which of the following the correct definition of “atomic actions”?
   a. Only one thread at a time is allowed to execute within a synchronized “critical section”
   b. Causes the current thread to wait until another thread notifies it to wake up
   c. Ensure that changes to a variable are always consistent & visible to other threads
   d. Concurrent calls will “busy wait”

2. Which of the following are the primary types of synchronization mechanisms provided by Java built-in monitor objects?
   a. Coordination
   b. Atomic actions
   c. Barrier synchronization
   d. Mutual exclusion