POLYTECH TOURS

Équipe OC
ERL-CNRS 6305

Laboratoire d'Informatique
EA 6300

Introduction to parallel computing

Chapter 3: concurrent programming in Java

Jorge E. Mendoza
Maître de Conférences
Polytech Tours
jorge.mendoza@univ-tours.fr

2014 - 2015

Acknowledgements

- Slides in this chapter are based on Lesson: concurrency of the Java Tutorials (http://docs.oracle.com/javase/tutorial/essential/concurrency/index.html). The code used in some of the examples comes from the same tutorial
- Some slides are taken from the tutorial: Java Concurrency Utilities by Jakob Jenkov (http://tutorials.jenkov.com/java-util-concurrent/index.html). Some code is also taken from this tutorial.

Introduction

- The Java platform is designed from the ground up to support concurrent programming, with basic concurrency support in the Java programming language and the Java class libraries
- Since version 5.0, the Java platform has also included high-level concurrency APIs
- The objective of this chapter are twofold:
  - Introducing Java platform's basic concurrency support and summarizing some of the high-level APIs in the `java.util.concurrent` packages
  - Discussing the most common synchronization problems and their most common solutions

Processes and Threads
A mind refresher

- In concurrent programming, there are two basic units of execution: ***processes*** and ***threads***
- In the Java programming language, concurrent programming is mostly concerned with threads. However, processes are also important
- A computer system normally has many active processes and threads.
- This is true even in systems that only have a single execution core, and thus only have one thread actually executing at any given moment
- Processing time for a single core is shared among processes and threads through an OS feature called time slicing (we discussed this in course on UNIX last semester, remember?)

Processes and Threads
Definitions: what is a Process (in the Java world, anyway)

- A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space
- Most implementations of the Java virtual machine run as a single process
- A Java application can create additional processes using a `ProcessBuilder` object (but multiprocess applications are beyond the scope of our course)

Processes and Threads

Definitions: what is a Thread (in the Java world, anyway)

- Threads are sometimes called *lightweight* processes
- Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process
- Threads exist within a process — every process has at least one
- Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication

Processes and Threads
Definitions: what is a Thread (in the Java world, anyway)

- Multithreaded execution is an essential feature of the Java platform
- Every application has at least one thread (or several, if you count "system" threads that do things like memory management and signal handling)
- From the application programmer's point of view, you start with just one thread, called the *main thread*
- This thread has the ability to create additional threads

Creating threads

- Each thread is associated with an instance of the class `Thread`. There are two basic strategies for using `Thread` objects to create a concurrent application:
  - To directly control thread creation and management, simply instantiate `Thread` each time the application needs to initiate an asynchronous task
  - To abstract thread management from the rest of your application, pass the application's tasks to an executor

Using Thread objects

- An application that creates an instance of `Thread` must provide the code that will run in that thread. There are two ways to do this:
  - Provide a `Runnable` object
    - The `Runnable` interface defines a single method, `run()`, meant to contain the code executed in the thread
    - The `Runnable` object is passed to the `Thread` constructor
  - Subclass `Thread`
    - The `Thread` class itself implements `Runnable`, though its run method does nothing
    - An application can subclass Thread, providing its own implementation of run

JAVA

polytech.tours.di.parallel.tutorial.HelloRunnable.java
polytech.tours.di.parallel.tutorial.HelloThread.java

Using Thread objects

- Which of these idioms should you use?
  - The first idiom, which employs a `Runnable` object, is more general, because the `Runnable` object can subclass a class other than `Thread`
  - The second idiom is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of `Thread`.
- We will focus on the first approach. Why?
  - This approach more flexible
  - It is applicable to the high-level thread management APIs that we will discuss later

Using Thread objects
Pausing execution with `sleep()`

- `Thread.sleep` causes the current thread to suspend execution for a specified period
- This is an efficient means of making processor time available to the other threads of an application or other applications that might be running on a computer system
- The `sleep()` method can also be used for pacing and waiting for another thread with duties that are understood to have time requirements
- Notice that in the example `main()` declares that it throws `InterruptedException`. This is an exception that sleep throws when another thread interrupts the current thread while sleep is active

Using Thread objects
Interrupts

- An interrupt is an indication to a thread that it should stop what it is doing and do something else
- It's up to the programmer to decide exactly how a thread responds to an interrupt, but it is very common for the thread to terminate
- A thread sends an interrupt by invoking `interrupt()` on the `Thread` object for the thread to be interrupted
- For the interrupt mechanism to work correctly, the interrupted thread must support its own interruption…

Using Thread objects

Interrupts: supporting interruption

- How does a thread support its own interruption?
- This depends on what it's currently doing
- If the thread is frequently invoking methods that throw `InterruptedException`, it simply returns from the `run()` method after it catches that exception

```
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]);
}
```

polytech.tours.di.parallel.tutorial.SleepMessagesInterrupt.java

- Many methods throwing `InterruptedException` are designed to cancel their operation and return immediately when an interrupt is received

Using Thread objects
Interrupts: supporting interruption

- What if a thread goes a long time without invoking a method that throws `InterruptedException`?
- Then it must periodically invoke `Thread.interrupted()`, which returns `true` if an interrupt has been received
- In more complex applications, it might make more sense to throw an `InterruptedException`

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

```
if (Thread.interrupted()) {
    throw new InterruptedException();
}
```

Using Thread objects
Interrupts: the Interrupt Status Flag

- The interrupt mechanism is implemented using an internal flag known as the ***interrupt status***
- Invoking `Thread.interrupt()` sets this flag
- When a thread checks for an interrupt by invoking the static method `Thread.interrupted()`, interrupt status is cleared
- The non-static `isInterrupted()` method, which is used by one thread to query the interrupt status of another, does not change the interrupt status flag
- By convention, any method that exits by throwing an `InterruptedException` clears interrupt status when it does so

Using Thread objects
Joins

- The `join()` method allows one thread to wait for the completion of another
- If t is a `Thread` object whose thread is currently executing, `t.join();` causes the current thread to pause execution until `t`'s thread terminates
- Overloads of `join()` allow the programmer to specify a waiting period. However, join is dependent on the OS for timing, so you should not assume that `join()` will wait exactly as long as you specify (BTW this is also the case for `sleep()`)
- Like `sleep()`, `join()` responds to an interrupt by exiting with an `InterruptedException`

Using Thread objects
Intermediate wrap-up example

polytech.tours.di.parallel.tutorial.SimpleThreads.java

Synchronization

- Threads communicate primarily by sharing access to fields and the objects reference fields refer to
- This form of communication is extremely efficient, but makes two kinds of errors possible: *thread interference* and *memory consistency errors*
- The tool needed to prevent these errors is *synchronization* (you already know this, right?)
- Synchronization can introduce thread contention, which occurs when two or more threads try to access the same resource simultaneously and cause the Java runtime to execute one or more threads more slowly, or even suspend their execution. Starvation and livelock are forms of thread contention

Synchronization
Thread interference

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

- `Counter` is designed so that each invocation of increment will add 1 to c, and each invocation of decrement will subtract 1 from c
- If a `Counter` object is referenced from multiple threads, interference between threads may prevent this from happening as expected
- Interference happens when two operations, running in different threads, but acting on the same data, *interleave*

Synchronization
Thread interference

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

- It might not seem possible for operations to interleave, since both operations on c are single, simple statements
- However, even simple statements can translate to multiple steps by the virtual machine
- The single expression c++ can be decomposed into three steps:
  - Retrieve the current value of c.
  - Increment the retrieved value by 1.
  - Store the incremented value back in c.

Synchronization
Thread interference

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

- Suppose `Thread` A invokes increment at about the same time `Thread` B invokes decrement
- If the initial value of c is 0, their interleaved actions might follow this sequence:
  - `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

Synchronization
Memory consistency errors

- *Memory consistency errors* occur when different threads have inconsistent views of what should be the same data
- The causes of memory consistency errors are complex (we will study them in the next chapter)
- By now let's just introduce some strategies to avoid them
- The key to avoiding memory consistency errors is understanding the *happens-before relationship*
- This relationship is simply a guarantee that memory writes by one specific statement are visible to another specific statement

Synchronization
Memory consistency errors

Example

Suppose a simple `int` field is defined and initialized:

```
int counter = 0;
```

The counter field is shared between two threads, A and B

Suppose thread A increments counter:

```
counter++;
```

Then, shortly afterwards, thread B prints out counter:

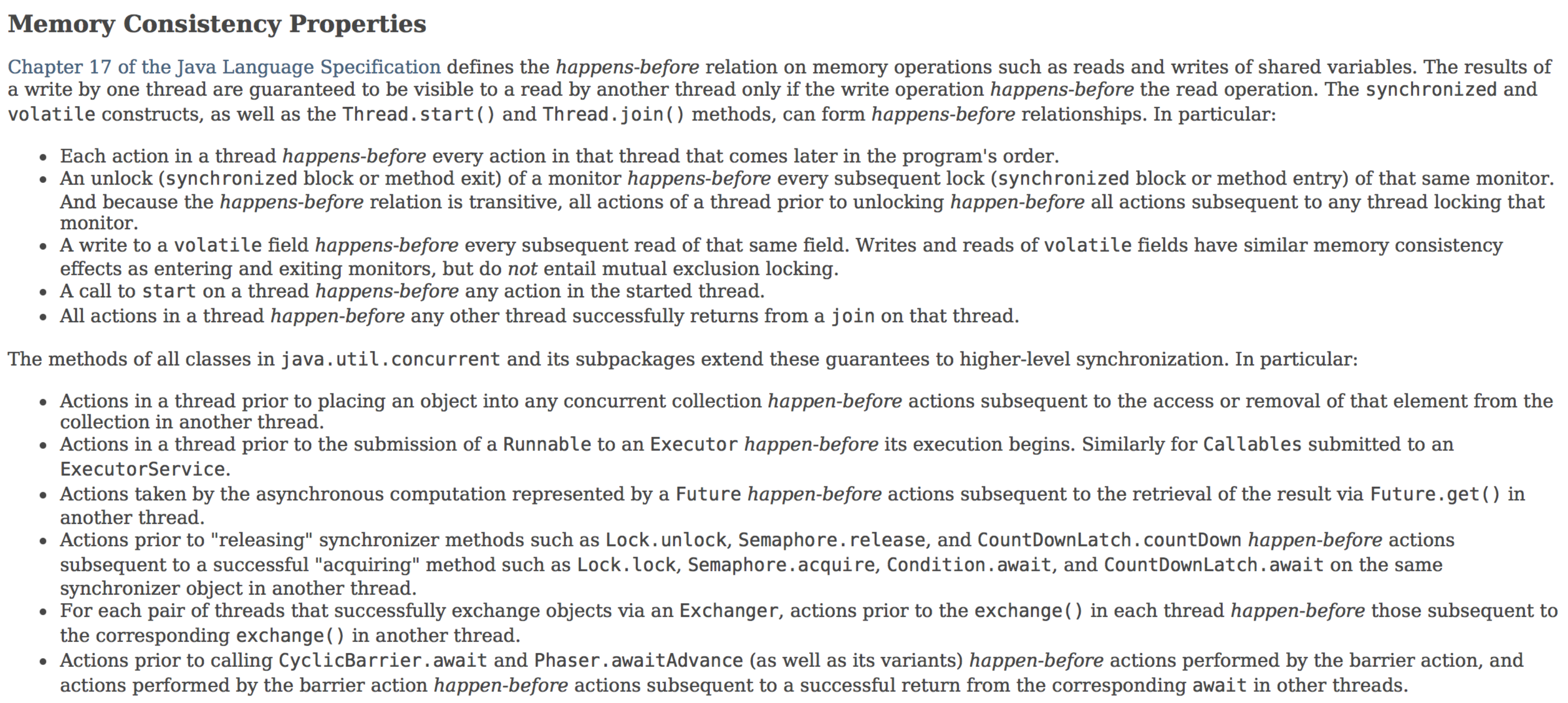
```
System.out.println(counter);
```

- If the two statements had been executed in the same thread, it would be safe to assume that the value printed out would be "1"
- If the two statements are executed in separate threads, the value printed out might well be "0" unless the programmer has established a happens-before relationship between these two statements

Synchronization
Memory consistency errors

- We've already seen two actions that create happens-before relationships
  - `Thread.start`: every statement that has a happens-before relationship with that statement also has a happens-before relationship with every statement executed by the new thread. The effects of the code that led up to the creation of the new thread are visible to the new thread
  - `Thread.join:` when a thread terminates and causes a `joint()` in another thread to return, then all the statements executed by the terminated thread have a happens-before relationship with all the statements following the successful join. The effects of the code in the thread are now visible to the thread that performed the join

Synchronization
Memory consistency errors: recommended reading



Memory Consistency Properties

Chapter 17 of the Java Language Specification defines the *happens-before* relation on memory operations such as reads and writes of shared variables. The results of a write by one thread are guaranteed to be visible to a read by another thread only if the write operation *happens-before* the read operation. The `synchronized` and `volatile` constructs, as well as the `Thread.start()` and `Thread.join()` methods, can form *happens-before* relationships. In particular:

- Each action in a thread *happens-before* every action in that thread that comes later in the program's order.
- An unlock (`synchronized` block or method exit) of a monitor *happens-before* every subsequent lock (`synchronized` block or method entry) of that same monitor. And because the *happens-before* relation is transitive, all actions of a thread prior to unlocking *happen-before* all actions subsequent to any thread locking that monitor.
- A write to a `volatile` field *happens-before* every subsequent read of that same field. Writes and reads of `volatile` fields have similar memory consistency effects as entering and exiting monitors, but do *not* entail mutual exclusion locking.
- A call to `start` on a thread *happens-before* any action in the started thread.
- All actions in a thread *happen-before* any other thread successfully returns from a `join` on that thread.

The methods of all classes in `java.util.concurrent` and its subpackages extend these guarantees to higher-level synchronization. In particular:

- Actions in a thread prior to placing an object into any concurrent collection *happen-before* actions subsequent to the access or removal of that element from the collection in another thread.
- Actions in a thread prior to the submission of a `Runnable` to an `Executor` *happen-before* its execution begins. Similarly for `Callables` submitted to an `ExecutorService`.
- Actions taken by the asynchronous computation represented by a `Future` *happen-before* actions subsequent to the retrieval of the result via `Future.get()` in another thread.
- Actions prior to "releasing" synchronizer methods such as `Lock.unlock`, `Semaphore.release`, and `CountDownLatch.countDown` *happen-before* actions subsequent to a successful "acquiring" method such as `Lock.lock`, `Semaphore.acquire`, `Condition.await`, and `CountDownLatch.await` on the same synchronizer object in another thread.
- For each pair of threads that successfully exchange objects via an `Exchanger`, actions prior to the `exchange()` in each thread *happen-before* those subsequent to the corresponding `exchange()` in another thread.
- Actions prior to calling `CyclicBarrier.await` and `Phaser.awaitAdvance` (as well as its variants) *happen-before* actions performed by the barrier action, and actions performed by the barrier action *happen-before* actions subsequent to a successful return from the corresponding `await` in other threads.

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/package-summary.html#MemoryVisibility

Synchronization
Synchronized methods

- The Java programming language provides two basic synchronization idioms: *synchronized methods* and *synchronized statements*
- Let’s take a look at synchronized methods first…

Synchronization
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;
    }
}
```

If count is an instance of `SynchronizedCounter`, making these methods synchronized has two effects:

- First, it is not possible for two invocations of synchronized methods on the same object to interleave
- Second, when a synchronized method exits, it automatically establishes a happens-before relationship with *any subsequent invocation* of a synchronized method for the same object

Synchronization
Synchronized methods

- When constructing an object that will be shared between threads, be very careful that a reference to the object does not "leak" prematurely
- For example, suppose you want to maintain a `List` called `instances` containing every instance of class
  - You might be tempted to add the following line to your constructor:

```
instances.add(this);
```

- But then other threads can use instances to access the object before construction of the object is complete
- BTW: constructors cannot be synchronized (why?)
- This strategy is simple and effective, but can present problems with liveness (we'll talk about this in a minute)

Synchronization
Intrinsic locks

- Synchronization is built around an internal entity known as the *intrinsic lock*
- Every object has an intrinsic lock associated with it
- A thread that needs exclusive and consistent access to an object's fields has to *acquire* the object's intrinsic lock before accessing them, and then *release* the intrinsic lock when it's done with them
- A thread is said to *own* the intrinsic lock between the time it has acquired the lock and released the lock
- As long as a thread owns an intrinsic lock, no other thread can acquire the same lock
- When a thread releases an intrinsic lock, a happens-before relationship is established between that action and any subsequent acquisition of the same lock

Synchronization
Locks and synchronized methods

- When a thread invokes a synchronized method, it automatically acquires the intrinsic lock for that method's object and releases it when the method returns
- The lock release occurs even if the return was caused by an uncaught exception
- What happens when a static synchronized method is invoked, since a static method is associated with a class, not an object?
- The thread acquires the intrinsic lock for the `Class` object associated with the class. Thus access to class's static fields is controlled by a lock that's distinct from the lock for any instance of the class

Synchronization
Synchronized statements

- Another way to create synchronized code is with *synchronized statements*
- Unlike synchronized methods, synchronized statements must specify the object that provides the intrinsic lock:

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

In this example, the `addName` method needs to synchronize changes to `lastName` and `nameCount`, but also needs to avoid synchronizing invocations of other objects' methods (Invoking other objects' methods from synchronized code can create problems that you will understand in a minute)

Synchronization
Synchronized statements

- Synchronized statements are also useful for improving concurrency with fine-grained synchronization...

Synchronization
Synchronized statements

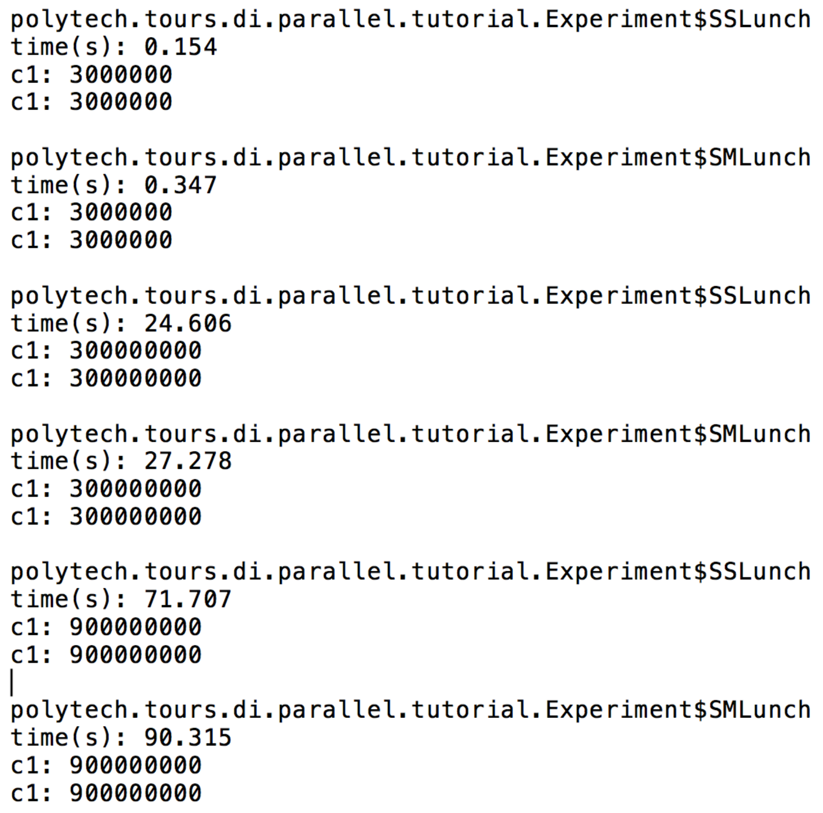
```
public class MsLunch {
    private long c1 = 0;
    private long c2 = 0;
    private Object lock1 = new Object();
    private Object lock2 = new Object();

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

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

- Suppose, for example, class `MsLunch` has two instance fields, `c1` and `c2`, that are never used together
- All updates of these fields must be synchronized, but there's no reason to prevent an update of `c1` from being interleaved with an update of `c2` — and doing so reduces concurrency by creating unnecessary blocking
- Instead of using synchronized methods or otherwise using the lock associated with `this`, we create two objects solely to provide locks

Synchronization

Synchronized statements vs. synchronized method: an experiment

polytech.tours.di.parallel.experiments.Experiment1.java



```
polytech.tours.di.parallel.tutorial.Experiment$SSLunch
time(s): 0.154
c1: 3000000
c1: 3000000

polytech.tours.di.parallel.tutorial.Experiment$SMLunch
time(s): 0.347
c1: 3000000
c1: 3000000

polytech.tours.di.parallel.tutorial.Experiment$SSLunch
time(s): 24.606
c1: 300000000
c1: 300000000

polytech.tours.di.parallel.tutorial.Experiment$SMLunch
time(s): 27.278
c1: 300000000
c1: 300000000

polytech.tours.di.parallel.tutorial.Experiment$SSLunch
time(s): 71.707
c1: 900000000
c1: 900000000

polytech.tours.di.parallel.tutorial.Experiment$SMLunch
time(s): 90.315
c1: 900000000
c1: 900000000
```

Synchronization
Reentrant synchronization

- Recall that a thread cannot acquire a lock owned by another thread
- But a thread can acquire a lock that it already owns
- Allowing a thread to acquire the same lock more than once enables *reentrant synchronization*
- This describes a situation where synchronized code, directly or indirectly, invokes a method that also contains synchronized code, and both sets of code use the same lock
- Without reentrant synchronization, synchronized code would have to take many additional precautions to avoid having a thread cause itself to block

Liveness

- Liveness is a concurrent application's ability to execute in a timely manner
- Let’s take a look at the most common kind of liveness problems:
  - Deadlocks
  - Starvation
  - Livelocks

Liveness
Deadlocks: a fable

- Alphonse and Gaston are friends, and great believers in courtesy
- A strict rule of courtesy is that when you bow to a friend, you must remain bowed until your friend has a chance to return the bow
- Unfortunately, this rule does not account for the possibility that two friends might bow to each other at the same time
- Lets take a look some code modeling this situation…

JAVA polytech.tours.di.parallel.tutorial.Deadlock.java

- When `Deadlock` runs, it's extremely likely that both threads will block when they attempt to invoke `bowBack()`. Neither block will ever end, because each thread is waiting for the other to exit bow

Liveness
Starvation

- *Starvation* describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress
- This happens when shared resources are made unavailable for long periods by "greedy" threads
- For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked
- Let's take a look at one simple example…

JAVA polytech.tours.di.parallel.tutorial.BankAccountStarvation.java

Liveness
Livelocks

- A thread often acts in response to the action of another thread
- If the other thread's action is also a response to the action of another thread, then *livelock* may result
- As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work
- This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphonse moves to his right, while Gaston moves to his left. They're still blocking each other, so...

Guarded blocks

- Threads often have to coordinate their actions
- The most common coordination idiom is the *guarded block*
- Such a block begins by polling a condition that must be true before the block can proceed
- There are a number of steps to follow in order to do this correctly…

Guarded blocks

- Suppose, for example `guardedJoy()` is a method that must not proceed until a shared variable joy has been set by another thread
- One could in theory just have the method loop until the condition is satisfied, but that loop is wasteful, since it executes continuously while waiting

```
public void guardedJoy() {
    // Simple loop guard. Wastes
    // processor time. Don't do this!
    while(!joy) {}
    System.out.println("Joy has been achieved!");
}
```

Guarded blocks

- Suppose, for example `guardedJoy()` is a method that must not proceed until a shared variable joy has been set by another thread
- One could in theory just have the method loop until the condition is satisfied, but that loop is wasteful, since it executes continuously while waiting
- A more efficient guard invokes `Object.wait()` to suspend the current thread (sounds familiar?)
- The invocation of `wait()` does not return until another thread has issued a notification that some special event may have occurred (though not necessarily the event this thread is waiting for, again, does it sound familiar?)

Guarded blocks

```
public synchronized void guardedJoy() {
    // This guard only loops once for each special event, which may not
    // be the event we're waiting for.
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}
```

Always invoke wait inside a loop that tests for the condition being waited for. Don't assume that the interrupt was for the particular condition you were waiting for, or that the condition is still true

Guarded blocks

```
public synchronized void guardedJoy() {
    // This guard only loops once for each special event, which may not
    // be the event we're waiting for.
    while(!joy) {
        try {
            wait();
        } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
}
```

Like many methods that suspend execution, wait can throw `InterruptedException`. In this example, we can just ignore that exception — we only care about the value of joy

Guarded blocks

- When `wait()` is invoked, the thread releases the lock and suspends execution. At some future time, another thread will acquire the same lock and invoke `Object.notifyAll()`, informing all threads waiting on that lock that something important has happened:

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

Tips

There is a second notification method, `notify()`, which wakes up a single thread. Because notify doesn't allow you to specify the thread that is woken up, it is useful only in massively parallel applications (programs with a large number of threads, all doing similar chores)

Guarded blocks

- Let’s take a look at guarded blocks in action…

Guarded blocks

- We are going to use guarded blocks to create a *Producer-Consumer* application
- This kind of application shares data between two threads:
  - The *producer* creates the data
  - The *consumer* does something with the data
- The two threads communicate using a shared object
- Coordination is essential: the consumer thread must not attempt to retrieve the data before the producer thread has delivered it, and the producer thread must not attempt to deliver new data if the consumer hasn't retrieved the old data

polytech.tours.di.parallel.tutorial.Producer.java

polytech.tours.di.parallel.tutorial.Drop.java

polytech.tours.di.parallel.tutorial.Consumer.java

polytech.tours.di.parallel.tutorial.ProducerConsumer.java

Guarded blocks

- We are going to use guarded blocks to create a *Producer-Consumer* application
- This kind of application shares data between two threads:

Tips

The Drop class was written in order to demonstrate guarded blocks. To avoid re-inventing the wheel, examine the existing data structures in the Java Collections Framework before trying to code your own data-sharing objects (we'll look at some of them later)

hasn't retrieved the old data

JAVA polytech.tours.di.parallel.tutorial.Producer.java
polytech.tours.di.parallel.tutorial.Drop.java

JAVA polytech.tours.di.parallel.tutorial.Consumer.java
polytech.tours.di.parallel.tutorial.ProducerConsumer.java

Immutable objects

- An object is considered *immutable* if its state cannot change after it is constructed. Maximum reliance on immutable objects is widely accepted as a sound strategy for creating simple, reliable code
- Immutable objects are particularly useful in concurrent applications. Since they cannot change state, they cannot be corrupted by thread interference or observed in an inconsistent state
- Programmers are often reluctant to employ immutable objects (because they worry about the cost of creating a new object as opposed to updating an existing one) but the benefits may offset this overhead

Immutable objects
A strategy for defining immutable objects

- Don't provide "setter" methods
- Make all fields final and private
- Don't allow subclasses to override methods (use final classes or factory design patterns)
- If the instance fields include references to mutable objects, don't allow those objects to be changed:
  - Don't provide methods that modify the mutable objects
  - Don't share references to the mutable objects
  - Never store references to external, mutable objects passed to the constructor (make copies)
  - Return copies of your internal mutable objects (don’t return the originals)

Immutable objects
A strategy for defining immutable objects: an example

polytech.tours.di.parallel.tutorial.SinchronizedRGB.java

polytech.tours.di.parallel.tutorial.ImmutableRGB.java

High level concurrency objects

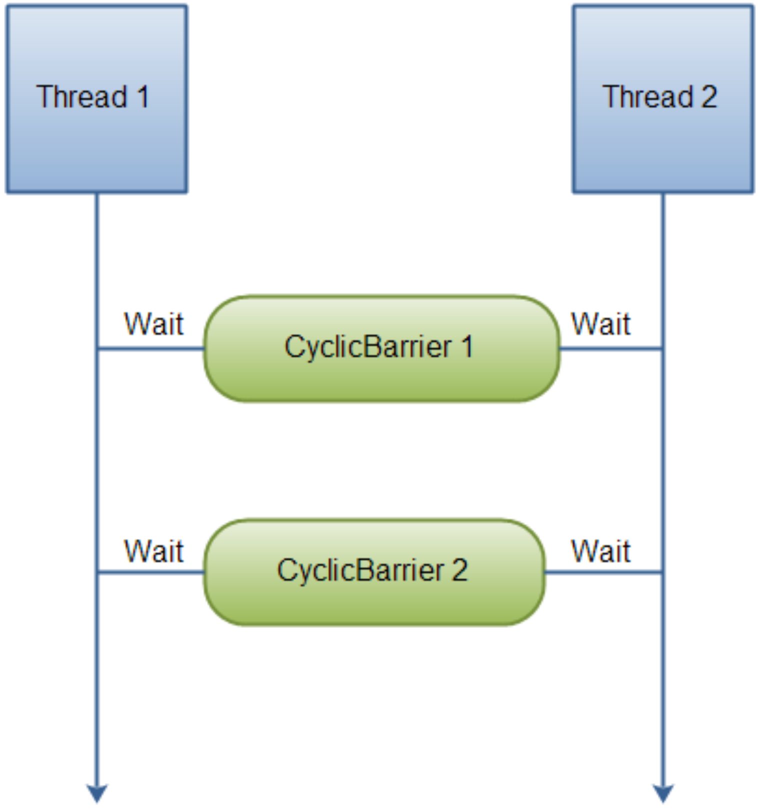
- So far, we have focused on the low-level APIs that have been part of the Java platform from the very beginning
- These APIs are adequate for very basic tasks, but higher-level building blocks are needed for more advanced tasks
- This is especially true for massively concurrent applications that fully exploit today's multiprocessor and multi-core systems

High level concurrency objects
Lock objects

- Synchronized code relies on a simple kind of reentrant lock
- This kind of lock is easy to use, but has many limitations
- More sophisticated locking idioms are supported by the `java.util.concurrent.locks` package
- We won't examine this package in detail (I expect you to do this at home), but instead will focus on its most basic interface, `Lock`

High level concurrency objects
`CyclicBarrier`

- The `CyclicBarrier` class is a synchronization mechanism that can synchronize threads progressing through some algorithm
- In other words, it is a barrier that all threads must wait at, until all threads reach it, before any of the threads can continue
- The threads wait for each other by calling the `await()` method on the `CyclicBarrier`. Once N threads are waiting at the `CyclicBarrier`, all threads are released and can continue running.



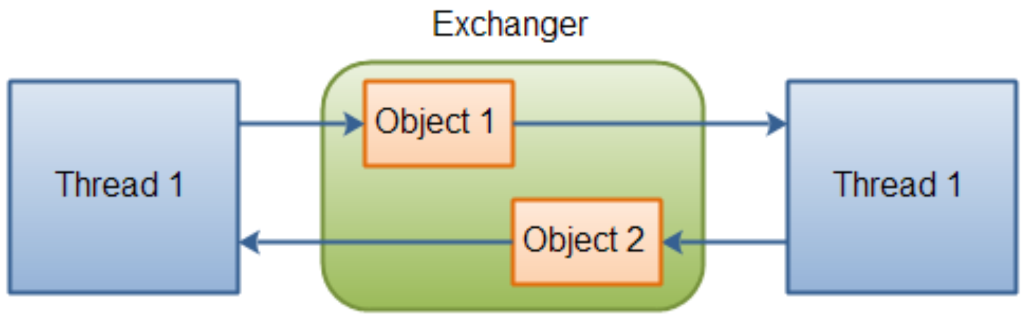
Source: Jenkov 2015

High level concurrency objects
`CyclicBarrier`: example

JAVA polytech.tours.di.parallel.tutorial.CyclicBarrierExample.java

High level concurrency objects
Exchanger

- The `java.util.concurrent.Exchanger` class represents a kind of rendezvous point where two threads can exchange objects



Source: Jenkov 2015

polytech.tours.di.parallel.tutorial.ExchangerExample.java

High level concurrency objects
Lock objects

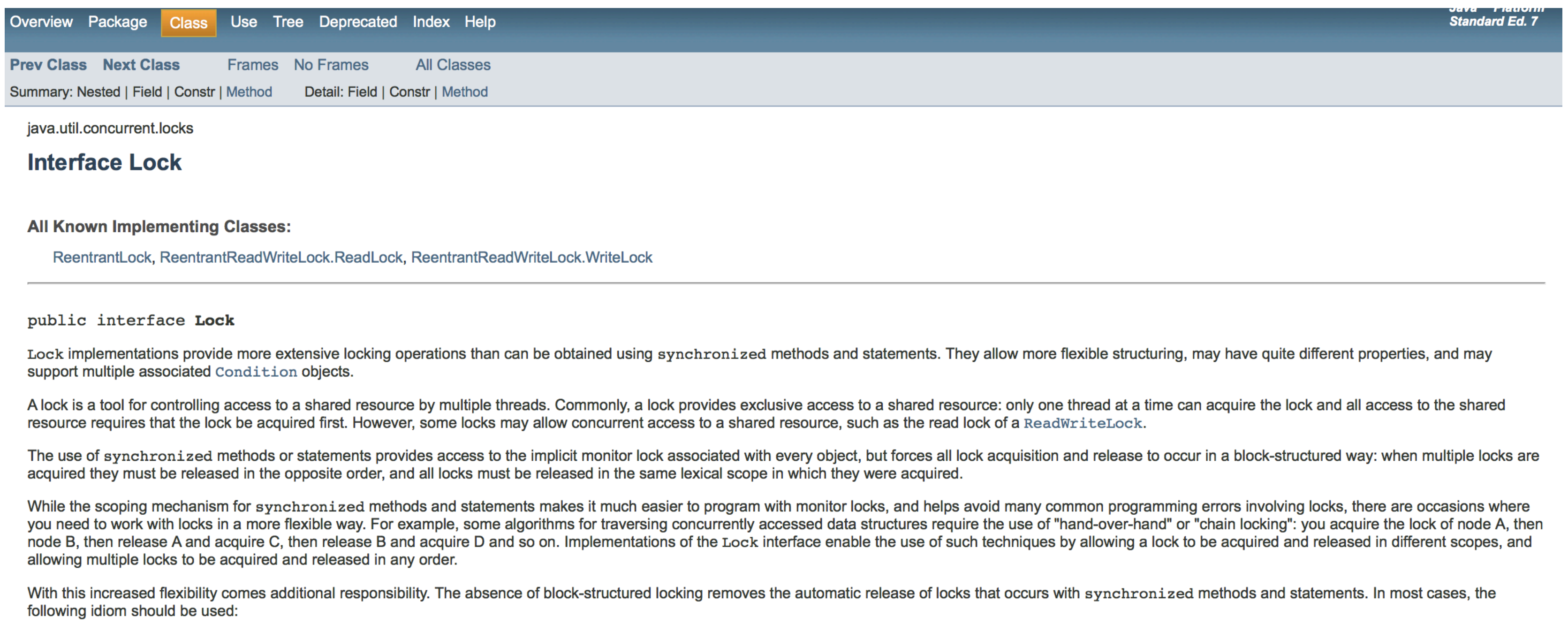
- `Lock` objects work very much like the implicit locks used by synchronized code. As with implicit locks, only one thread can own a `Lock` object at a time.
- `Lock` objects also support a wait/notify mechanism, through their associated `Condition` objects
- The biggest advantage of `Lock` objects over implicit locks is their ability to back out of an attempt to acquire a lock:
  - `tryLock()` backs out if the lock is not available immediately or before a timeout expires (if specified)
  - `lockInterruptibly()` method backs out if another thread sends an interrupt before the lock is acquired

High level concurrency objects
Lock objects: an example

- Remember our friends Gaston and Alphonse?
- Let's use **Lock** objects to solve the deadlock problem we saw in that example…

polytech.tours.di.parallel.tutorial.Safelock.java

High level concurrency objects
Lock objects: recommended reading


Overview Package Class Use Tree Deprecated Index Help
Java™ Platform Standard Ed. 7
Prev Class Next Class Frames No Frames All Classes
Summary: Nested | Field | Constr | Method Detail: Field | Constr | Method

java.util.concurrent.locks

Interface Lock

All Known Implementing Classes:

ReentrantLock, ReentrantReadWriteLock.ReadLock, ReentrantReadWriteLock.WriteLock

`public interface` **`Lock`**

`Lock` implementations provide more extensive locking operations than can be obtained using `synchronized` methods and statements. They allow more flexible structuring, may have quite different properties, and may support multiple associated `Condition` objects.

A lock is a tool for controlling access to a shared resource by multiple threads. Commonly, a lock provides exclusive access to a shared resource: only one thread at a time can acquire the lock and all access to the shared resource requires that the lock be acquired first. However, some locks may allow concurrent access to a shared resource, such as the read lock of a `ReadWriteLock`.

The use of `synchronized` methods or statements provides access to the implicit monitor lock associated with every object, but forces all lock acquisition and release to occur in a block-structured way: when multiple locks are acquired they must be released in the opposite order, and all locks must be released in the same lexical scope in which they were acquired.

While the scoping mechanism for `synchronized` methods and statements makes it much easier to program with monitor locks, and helps avoid many common programming errors involving locks, there are occasions where you need to work with locks in a more flexible way. For example, some algorithms for traversing concurrently accessed data structures require the use of "hand-over-hand" or "chain locking": you acquire the lock of node A, then node B, then release A and acquire C, then release B and acquire D and so on. Implementations of the `Lock` interface enable the use of such techniques by allowing a lock to be acquired and released in different scopes, and allowing multiple locks to be acquired and released in any order.

With this increased flexibility comes additional responsibility. The absence of block-structured locking removes the automatic release of locks that occurs with `synchronized` methods and statements. In most cases, the following idiom should be used:

http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/Lock.html

High level concurrency objects
`ReadWriteLock` objects

- A `ReadWriteLock` is an advanced thread lock mechanism
- It allows multiple threads to read a certain resource, but only one to write it, at a time
- The rules by which a thread is allowed to lock the `ReadWriteLock` either for reading or writing the guarded resource, are as follows:
  - **Read Lock:** If no threads have locked the `ReadWriteLock` for writing, and no thread have requested a write lock (but not yet obtained it). Thus, multiple threads can lock the lock for reading
  - **Write Lock:** If no threads are reading or writing. Thus, only one thread at a time can lock the lock for writing

High level concurrency objects
`ReadWriteLock` objects: example

```
ReadWriteLock readWriteLock = new ReentrantReadWriteLock();


readWriteLock.readLock().lock();

    // multiple readers can enter this section
    // if not locked for writing, and not writers waiting
    // to lock for writing.

readWriteLock.readLock().unlock();


readWriteLock.writeLock().lock();

    // only one writer can enter this section,
    // and only if no threads are currently reading.

readWriteLock.writeLock().unlock();
```

High level concurrency objects

Semaphore objects

- The `java.util.concurrent.Semaphore` class is a counting semaphore. That means that it has two main methods: `acquire() and release()`
- The counting semaphore is initialized with a given number of "permits"
- For each call to `acquire()` a permit is taken by the calling thread. For each call to `release()` a permit is returned to the semaphore
- At most N threads can pass the `acquire()` method without any `release()` calls, where N is the number of permits the semaphore was initialized with
- The permits are just a simple counter. Nothing fancy here.

High level concurrency objects
Semaphore objects

- As semaphore typically has two uses:
  - To guard a critical section against entry by more than N threads at a time
  - To send signals between two threads

High level concurrency objects
Semaphore objects

- If you use a semaphore to guard a critical section, the thread trying to enter the critical section will typically first try to acquire a permit, enter the critical section, and then release the permit again after:

```
Semaphore semaphore = new Semaphore(1);
//critical section
semaphore.acquire();
...
semaphore.release();
```

High level concurrency objects
Semaphore objects

- If you use a semaphore to send signals between threads, then you would typically have one thread call the `acquire()` method, and the other thread to call the `release()` method
- If no permits are available, the `acquire()` call will block until a permit is released by another thread. Similarly, a `release()` calls is blocked if no more permits can be released into this semaphore
- For instance, if `acquire()` was called after `Thread 1` had inserted an object in a shared list, and `Thread 2` had called `release()` just before taking an object from that list, you had essentially created a blocking queue (we'll see an example in the practical sessions)
- To guarantee fairness, use the appropriated constuctor

High level concurrency objects
Executors

- In all of the previous examples, there's a close connection between the task being done by a new thread, as defined by its `Runnable` object, and the thread itself, as defined by a `Thread` object
- This works well for small applications, but in large-scale applications, it makes sense to separate thread management and creation from the rest of the application
- Objects that encapsulate these functions are known as *executors*
- Let’s take a look…

High level concurrency objects
Executors interfaces

- The `java.util.concurrent` package defines three executor interfaces:
  - `Executor`: a simple interface that supports launching new tasks
  - `ExecutorService`: a subinterface of `Executor`, which adds features that help manage the lifecycle, both of the individual tasks and of the executor itself
  - `ScheduledExecutorService`: a subinterface of `ExecutorService`, supports future and/or periodic execution of tasks
- Recommended reading: the Javadoc of these three interfaces

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ScheduledExecutorService.html

High level concurrency objects
The `Executor` interface

- The `Executor` interface provides a single method, `execute()`, designed to be a drop-in replacement for a common thread-creation idiom
- If `r` is a `Runnable` object, and `e` is an `Executor` object you can replace

  ```
  (new Thread(r)).start();
  ```

  with

  ```
  e.execute(r);
  ```

- The low-level idiom creates a new thread and launches it immediately
- Depending on the `Executor` implementation, execute may do the same thing, but is more likely to use an existing worker thread to run `r`, or to place `r` in a queue to wait for a worker thread to become available

High level concurrency objects
The `ExecutorService` interface

- The `ExecutorService` interface supplements `execute()` with a similar, but more versatile `submit()` method
- Like `execute()`, `submit()` accepts `Runnable` objects, but also accepts `Callable` objects, which allow the task to return a value
- The `submit()` method returns a `Future` object, which is used to retrieve the `Callable` return value and to manage the status of both `Callable` and `Runnable` tasks
- `ExecutorService` also provides methods for submitting large collections of `Callable` objects
- `ExecutorService` also provides a number of methods for managing the shutdown of the executor. To support immediate shutdown, tasks should handle interrupts correctly

High level concurrency objects
The `ScheduledExecutorService` interface

- The `ScheduledExecutorService` interface supplements the methods of its parent `ExecutorService` with `schedule()`, which executes a `Runnable` or `Callable` task after a specified delay
- In addition, the interface defines `scheduleAtFixedRate` and `scheduleWithFixedDelay`, which executes specified tasks repeatedly, at defined intervals

High level concurrency objects
Thread pools

- Most of the executor implementations in `java.util.concurrent` use *thread pools*, which consist of *worker threads*
- This kind of thread exists separately from the `Runnable` and `Callable` tasks it executes and is often used to execute multiple tasks
- Using worker threads minimizes the overhead due to thread creation
- `Thread` objects use a significant amount of memory, and in a large-scale application, allocating and deallocating many thread objects creates a significant memory management overhead

High level concurrency objects
Thread pools

- One common type of thread pool is the *fixed thread pool*
- This type of pool always has a specified number of threads running; if a thread is somehow terminated while it is still in use, it is automatically replaced with a new thread
- Tasks are submitted to the pool via an internal queue, which holds extra tasks whenever there are more active tasks than threads
- An important advantage of the fixed thread pool is that applications using it *degrade gracefully* (i.e., the application does not service tasks as quickly as they come in but as as quickly as the system can sustain)

High level concurrency objects
Thread pools

- A simple way to create an executor that uses a fixed thread pool is to invoke the `newFixedThreadPool()` factory method in `java.util.concurrent.Executors` This class also provides the following factory methods:
  - `newCachedThreadPool():` creates an executor with an expandable thread pool. This executor is suitable for applications that launch many short-lived tasks
  - `newSingleThreadExecutor():`creates an executor that executes a single task at a time
- Several factory methods are `ScheduledExecutorService` versions of the above executors
- If none of the executors provided by the above factory methods meet your needs, constructing instances of `java.util.concurrent.ThreadPoolExecutor` or `java.util.concurrent.ScheduledThreadPoolExecutor` will give you additional options (please read the Java doc at home)

High level concurrency objects
Thread pools: an example

polytech.tours.di.parallel.tutorial.ThreadPooling.java

polytech.tours.di.parallel.tutorial.Sums.java

High level concurrency objects

Fork/Joint

- The fork/join framework is an implementation of the `ExecutorService` interface that helps you take advantage of multiple processors
- It is designed for work that can be broken into smaller pieces recursively
- As with any `ExecutorService` implementation, the fork/join framework distributes tasks to worker threads in a thread pool
- The fork/join framework is distinct because it uses a work-stealing algorithm: worker threads that run out of things to do can “steal” tasks from other threads that are still busy
- The center of the fork/join framework is the `ForkJoinPool`; it implements the core work-stealing algorithm and can execute `ForkJoinTask` processes

High level concurrency objects
Fork/Joint

- `ForkJoinTask` objects feature two specific methods:
  - The `fork()` method allows a `ForkJoinTask` to be planned for asynchronous execution. This allows a new `ForkJoinTask` to be launched from an existing one
  - The `join()` method allows a `ForkJoinTask` to wait for the completion of another one

The fork() and join() method names should not be confused with their POSIX counterparts with which a process can duplicate itself (remember UNIX?). Here, fork() only schedules a new task within a ForkJoinPool, but no child Java Virtual Machine is ever created

High level concurrency objects
Fork/Joint

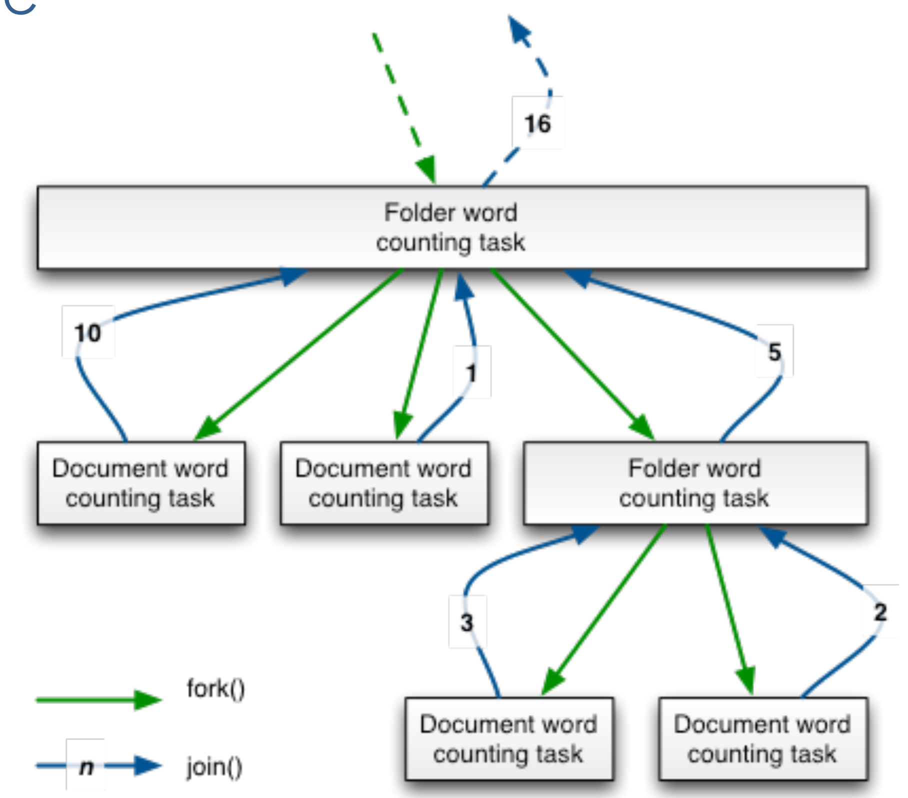
- There are two types of `ForkJoinTask` specializations:
  - Instances of `RecursiveAction` represent executions that do not yield a return value
  - Instances of `RecursiveTask` yield return values
- In general, `RecursiveTask` is preferred because most divide-and-conquer algorithms return a value from a computation over a data set
- For the execution of tasks, different synchronous and asynchronous options are provided, making it possible to implement elaborate patterns

High level concurrency objects
Fork/Joint: example

polytech.tours.di.parallel.tutorial.WordCounter.java

High level concurrency objects

Fork/Joint: example



16
Folder word counting task
10
1
5
Document word counting task
Document word counting task
Folder word counting task
3
2
fork()
n
join()
Document word counting task
Document word counting task

Source: Ponge 2011

High level concurrency objects
Fork/Joint: the strategy

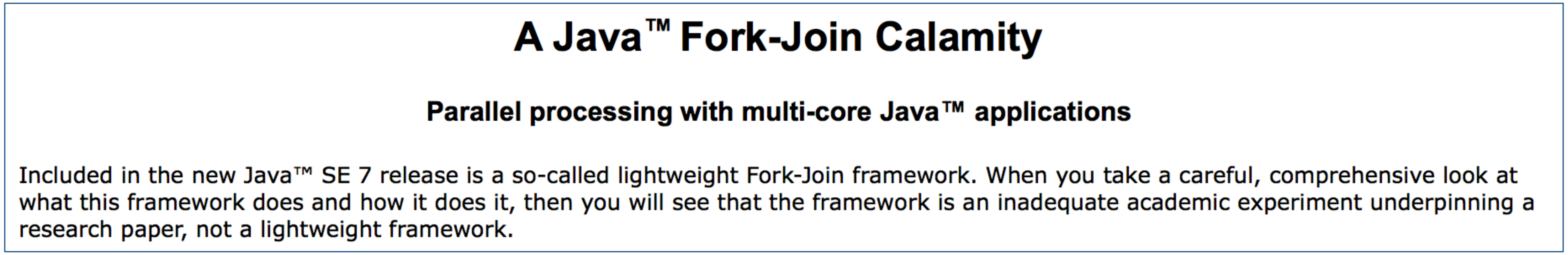
```
// PSEUDOCODE
Result solve(Problem problem) {
    if (problem.size < SEQUENTIAL_THRESHOLD)
        return solveSequentially(problem);
    else {
        Result left, right;
        INVOKE-IN-PARALLEL {
            left = solve(extractLeftHalf(problem));
            right = solve(extractRightHalf(problem));
        }
        return combine(left, right);
    }
}
```

Source: Goetz 2007

High level concurrency objects
Fork/Joint

- Some generally useful features in Java SE which are already implemented using the fork/join framework
- One such implementation, introduced in Java SE 8, is used by the `java.util.Arrays` class for its `parallelSort()` methods
- These methods are similar to sort(), but leverage concurrency via the fork/join framework
- Parallel sorting of large arrays is faster than sequential sorting when run on multiprocessor systems (we'll take a look that some of these in the practical sessions)

High level concurrency objects
Fork/Joint: recommended reading



A Java™ Fork-Join Calamity

Parallel processing with multi-core Java™ applications

Included in the new Java™ SE 7 release is a so-called lightweight Fork-Join framework. When you take a careful, comprehensive look at what this framework does and how it does it, then you will see that the framework is an inadequate academic experiment underpinning a research paper, not a lightweight framework.

http://coopsoft.com/ar/CalamityArticle.html

High level concurrency objects
Concurrent collections

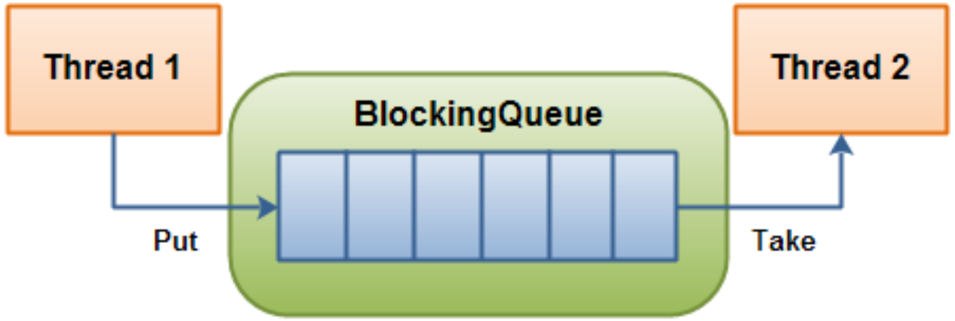
- The `java.util.concurrent` package includes a number of additions to the Java `Collections` Framework. These are most easily categorized by the collection interfaces provided:
  - `BlockingQueue`: defines a first-in-first-out data structure that blocks or times out when you attempt to add to a full queue, or retrieve from an empty queue
  - `ConcurrentMap`: is a subinterface of `java.util.Map` that defines useful atomic operations. These operations remove or replace a key-value pair only if the key is present, or add a key-value pair only if the key is absent (this helps avoiding synchronization)
  - The standard general-purpose implementation of `ConcurrentMap` is `ConcurrentHashMap`, which is a concurrent analog of `HashMap`

High level concurrency objects
Concurrent collections

- The `java.util.concurrent` package includes a number of additions to the Java `Collections` Framework. These are most easily categorized by the collection interfaces provided:
  - `ConcurrentNavigableMap` is a subinterface of `ConcurrentMap` that supports approximate matches. The standard general-purpose implementation of `ConcurrentNavigableMap` is `ConcurrentSkipListMap`, which is a concurrent analog of `TreeMap`
- All of these collections help avoid memory consistency errors by defining a happens-before relationship between an operation that adds an object to the collection with subsequent operations that access or remove that object

High level concurrency objects
Concurrent collections: example – blocking queue

- A `BlockingQueue` is typically used to have on thread produce objects, which another thread consumes



Source: Jenkov 2015

polytech.tours.di.parallel.tutorial.BlockingQueueExample.java

High level concurrency objects
Atomic variables

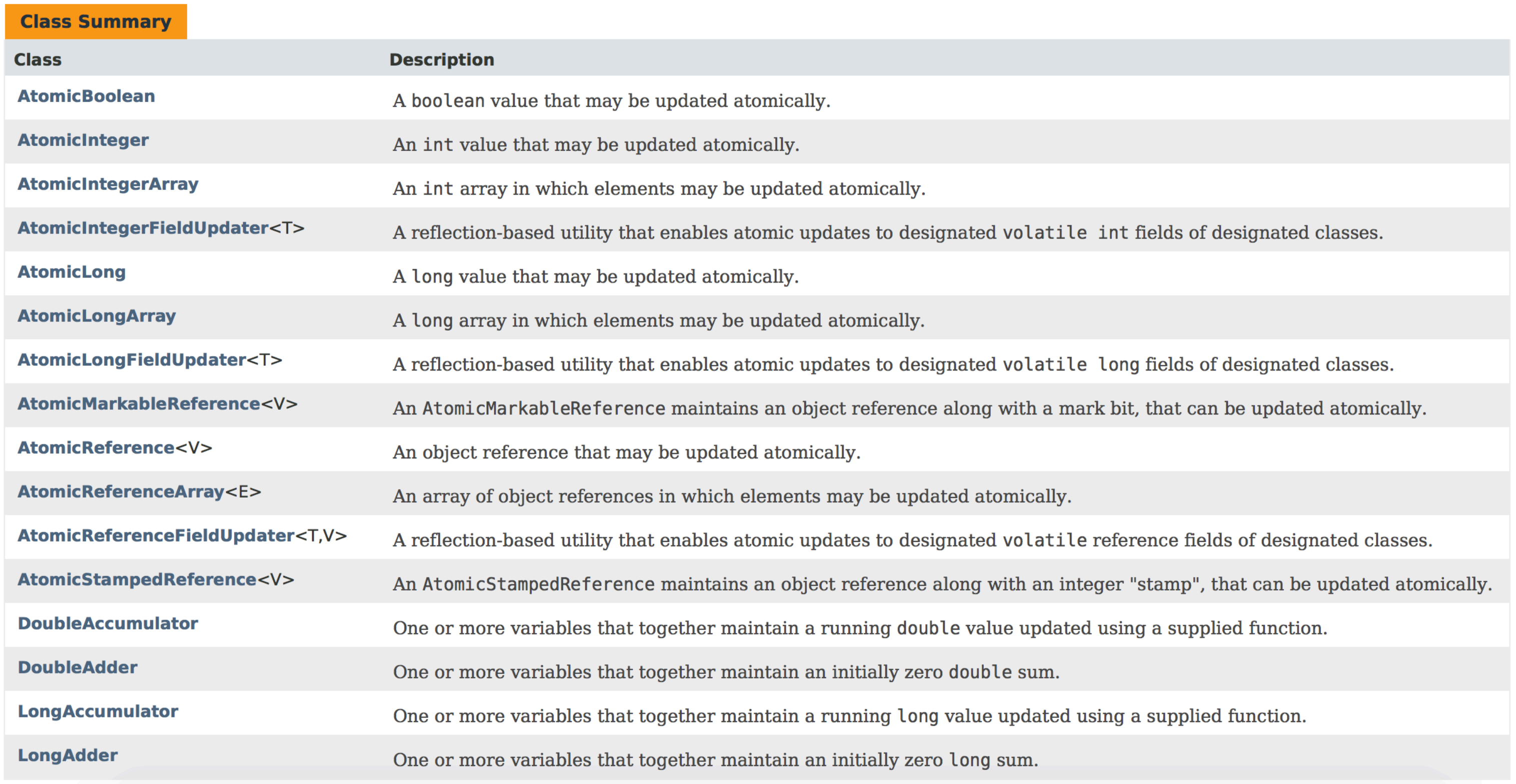
- The `java.util.concurrent.atomic` package defines classes that support atomic operations on single variables
- All classes have get and set methods that work like reads and writes on volatile variables
- That is, a set has a happens-before relationship with any subsequent get on the same variable
- The atomic `compareAndSet` method also has these memory consistency features, as do the simple atomic arithmetic methods that apply to integer atomic variables
- To see how this package might be used, let's return to the `Counter` class we originally used to demonstrate thread interference…

High level concurrency objects
Atomic variables: example

```
class Counter {
    private int c = 0;

    public void increment() {
        c++;
    }

    public void decrement() {
        c--;
    }

    public int value() {
        return c;
    }

}
```

```
class AtomicCounter {
    private AtomicInteger c = new AtomicInteger(0);

    public void increment() {
        c.incrementAndGet();
    }

    public void decrement() {
        c.decrementAndGet();
    }

    public int value() {
        return c.get();
    }

}
```

High level concurrency objects
Atomic variables: recommended reading



Class Summary

Class	Description
AtomicBoolean	A `boolean` value that may be updated atomically.
AtomicInteger	An `int` value that may be updated atomically.
AtomicIntegerArray	An `int` array in which elements may be updated atomically.
AtomicIntegerFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated `volatile int` fields of designated classes.
AtomicLong	A `long` value that may be updated atomically.
AtomicLongArray	A `long` array in which elements may be updated atomically.
AtomicLongFieldUpdater<T>	A reflection-based utility that enables atomic updates to designated `volatile long` fields of designated classes.
AtomicMarkableReference<V>	An `AtomicMarkableReference` maintains an object reference along with a mark bit, that can be updated atomically.
AtomicReference<V>	An object reference that may be updated atomically.
AtomicReferenceArray<E>	An array of object references in which elements may be updated atomically.
AtomicReferenceFieldUpdater<T,V>	A reflection-based utility that enables atomic updates to designated `volatile` reference fields of designated classes.
AtomicStampedReference<V>	An `AtomicStampedReference` maintains an object reference along with an integer "stamp", that can be updated atomically.
DoubleAccumulator	One or more variables that together maintain a running `double` value updated using a supplied function.
DoubleAdder	One or more variables that together maintain an initially zero `double` sum.
LongAccumulator	One or more variables that together maintain a running `long` value updated using a supplied function.
LongAdder	One or more variables that together maintain an initially zero `long` sum.

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/atomic/package-summary.html

High level concurrency objects
Concurrent random numbers

- Since JDK 7, `java.util.concurrent` includes a convenience class, `ThreadLocalRandom`, for applications that expect to use random numbers from multiple threads or `ForkJoinTasks`
- For concurrent access, using `ThreadLocalRandom` instead of `Math.random()` results in less contention and, ultimately, better performance
- All you need to do is call `ThreadLocalRandom.current()`, then call one of its methods to retrieve a random number.
- Here is one example:

```
    int r = ThreadLocalRandom.current().nextInt(4, 77);
```