Concurrency: Synchronization

Tutorial 14 (26 May 2021)

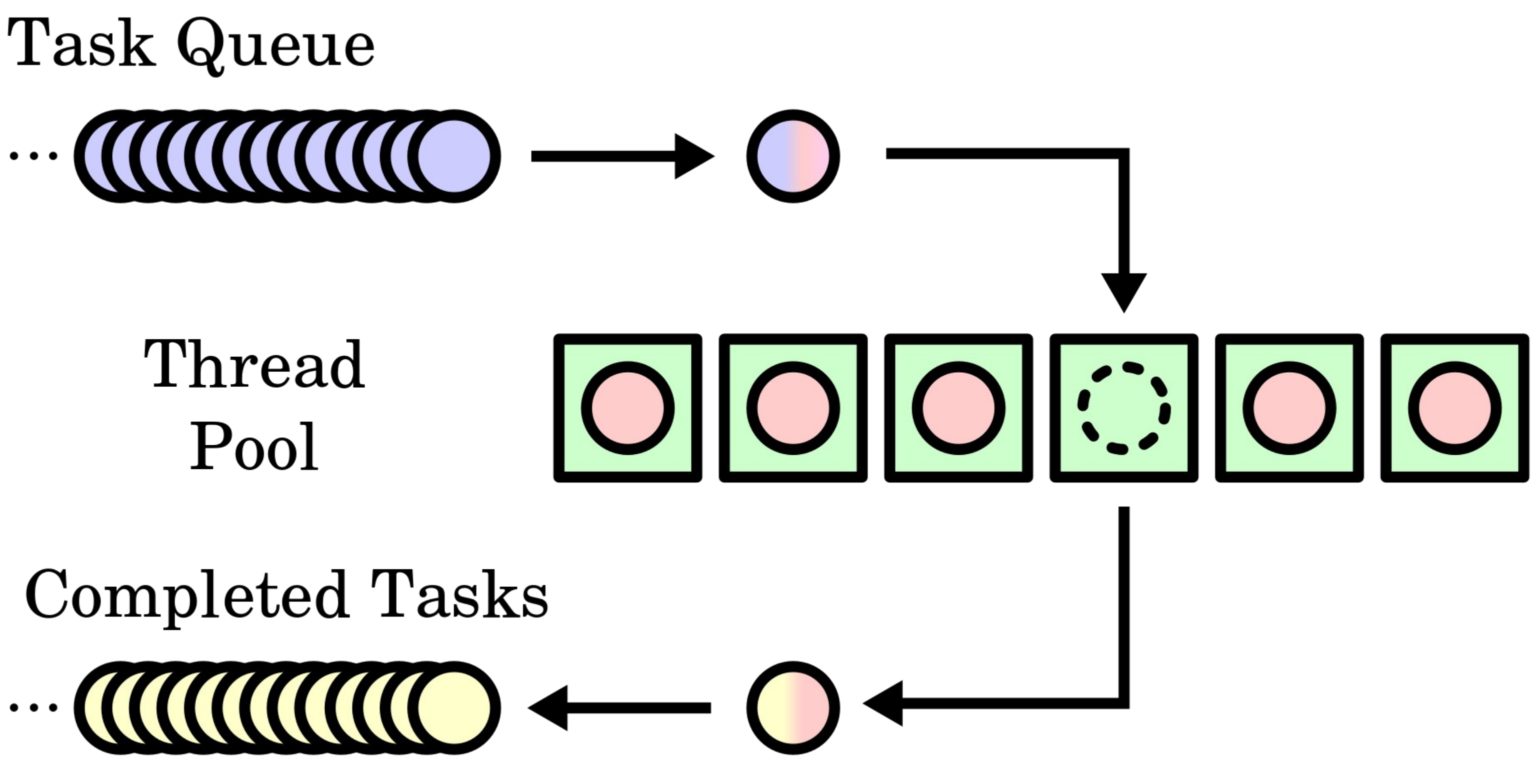
Daniël Kuijsters

WE START AT 8:30

Radboud University

Creating a thread is NOT for free!

Thread Pool



ExecutorService

- Interface for implementing a **thread pool**.
- Comes with lifecycle management methods.

```
public interface Executor {
    void execute(Runnable command);
}

public interface ExecutorService extends Executor {
    void shutdown();
    List<Runnable> shutdownNow();
    boolean isShutdown();
    boolean isTerminated();
    boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException;
    // … additional convenience methods for task submission
}
```

ExecutorService (2)

- For a Runnable r, instead of:

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

- Use:

```
ExecutorService exec = Executors.newCachedThreadPool();
// You only need to create exec once of course :-)
exec.execute(r);
```

ExecutorService (3)

- If you do NOT shut down the executor, then the program will keep running until the threads time out. In case of a CachedThreadPool, this will take 60 seconds.
- Hence, do NOT forget to shut down your executor!
- Calling execute after a shutdown(Now) will throw a RejectedExecutionException!

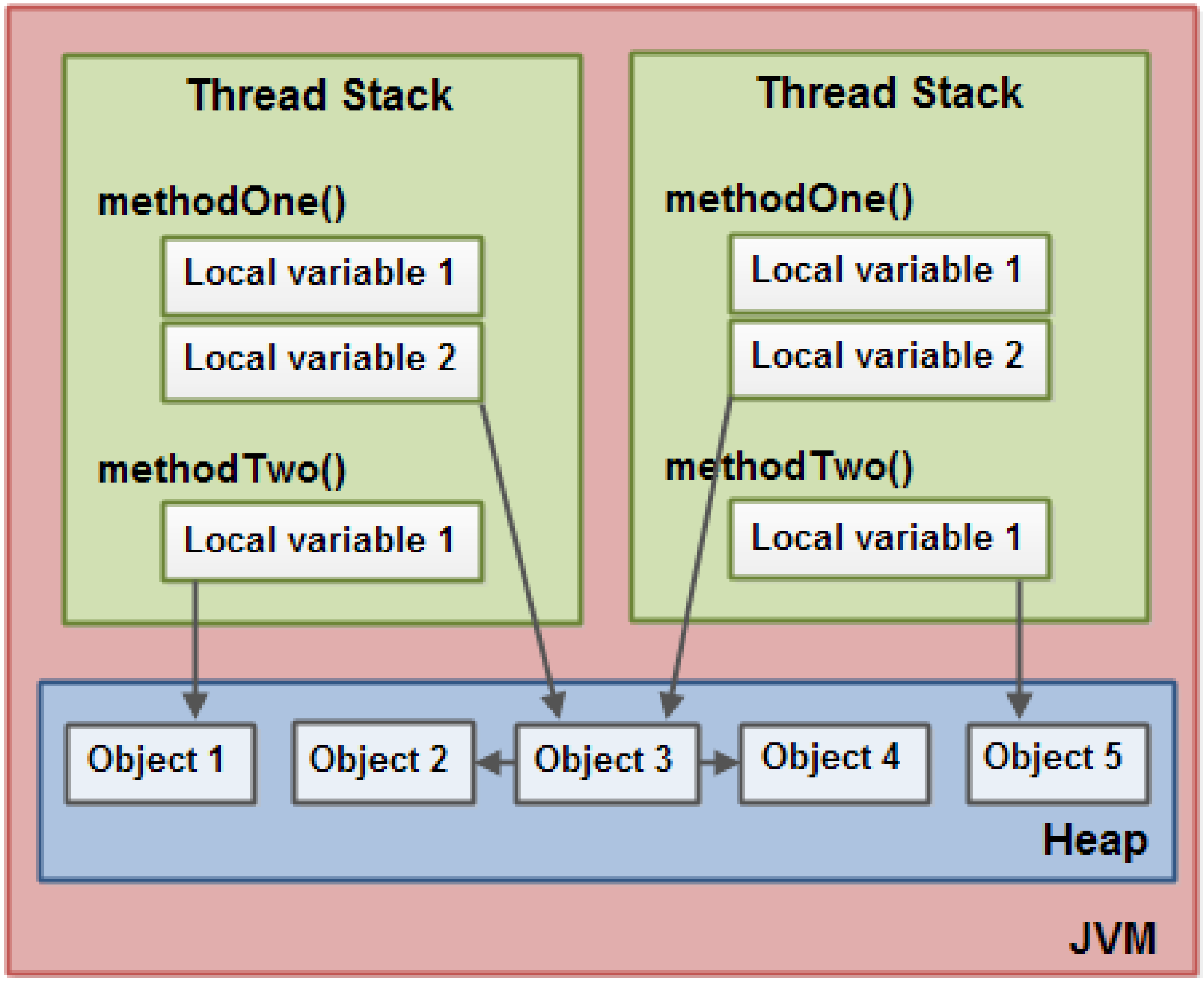
ExecutorService (4)

In order to shut down an executor:

```
public void shutdownAndAwaitTermination(ExecutorService exec) {
    exec.shutdown(); // Disable new tasks from being submitted
    try {
        if (!exec.awaitTermination(1, TimeUnit.MINUTES)) {
            // Cancel currently executing tasks
            List<Runnable> tasksNeverStarted = exec.shutdownNow();
            // Wait a while for tasks to respond to being canceled
            if (!exec.awaitTermination(1, TimeUnit.MINUTES)) {
                System.err.println("Exec did not terminate");
            }
        }
    } catch (InterruptedException e) {
        // (Re-)Cancel if current thread also interrupted
        List<Runnable> tasksNeverStarted = exec.shutdownNow();
        // Preserve interrupt status
        Thread.currentThread().interrupt();
    }
}
```

Demo: ExecutorService applied to FileFinder

Recall: JVM Memory Model



Race Conditions

A **race condition** occurs when the correctness of the computation depends on the relative timing or interleaving of multiple threads by the runtime.

Read-modify-write:

```
public class UnsafeSequence {
	private int value;

	public int getNext() {
		return value++;
	}
}
```

Check-then-act issue:

```
if (doing_this_is_allowed) {
	do_it();
}
```

Potential security vulnerability!

Why Is value++ Not Atomic?

```
public class UnsafeSequence {
    private int value;

    public int getNext() {
        return value++;
    }
}
```

```
$ javac UnsafeSequence.java
$ javap -c UnsafeSequence.class
```

```
public int getNext();
  Code:
     0: aload_0
     1: dup
     2: getfield     #2
     5: dup_x1
     6: iconst_1
     7: iadd
     8: putfield     #2
    11: ireturn
```

Critical Sections

- A code segment that may access (and update) data that is shared with at least one other thread.
- When a thread is executing in its critical section, no other thread is allowed to execute in its critical section → mutual exclusion.
- How can you guarantee this?

Monitor Locks / Intrinsic Locks

- Each Java object can act as a **lock** because associated to it is a so-called monitor lock / intrinsic lock.
- Only one thread can own the lock at the same time.
- Provides mutual exclusion.
- Use the **synchronized** keyword.

Synchronized Keyword

- A critical section is guarded as follows:

```
synchronized (object reference) {
    statements of critical section;
}
```

- Example:

```
synchronized (account) {
    account.deposit(1);
}
```

Synchronized Keyword (2)

- To guard the body of a method:

```
public void xMethod() {
	synchronized (this) {
		// method body
	}
}
```

- Equivalently:

```
public synchronized void xMethod() {
	// method body
}
```

Reentrancy

- When a thread requests a lock that is held by another thread, the requesting thread blocks.
- But what happens if a thread requests a lock that it already holds?
- Intrinsic locks **(synchronized)** are **reentrant**: the locks are acquired on a per-thread basis rather than per-invocation basis.
- It is implemented by associating with each lock an acquisition count and an owning thread.

What Would Go Wrong Here Without Reentrancy?

```java
public class Widget {
	public synchronized void doSomething() {
		// ...
	}
}

public class LoggingWidget extends Widget {
	public synchronized void doSomething() {
		System.out.println(toString() + ": calling doSomething");
		super.doSomething();
	}
}
```

Deadlock

- No thread can proceed because each thread is waiting for another to do some work first.
- This thread is waiting for lock to be unlocked

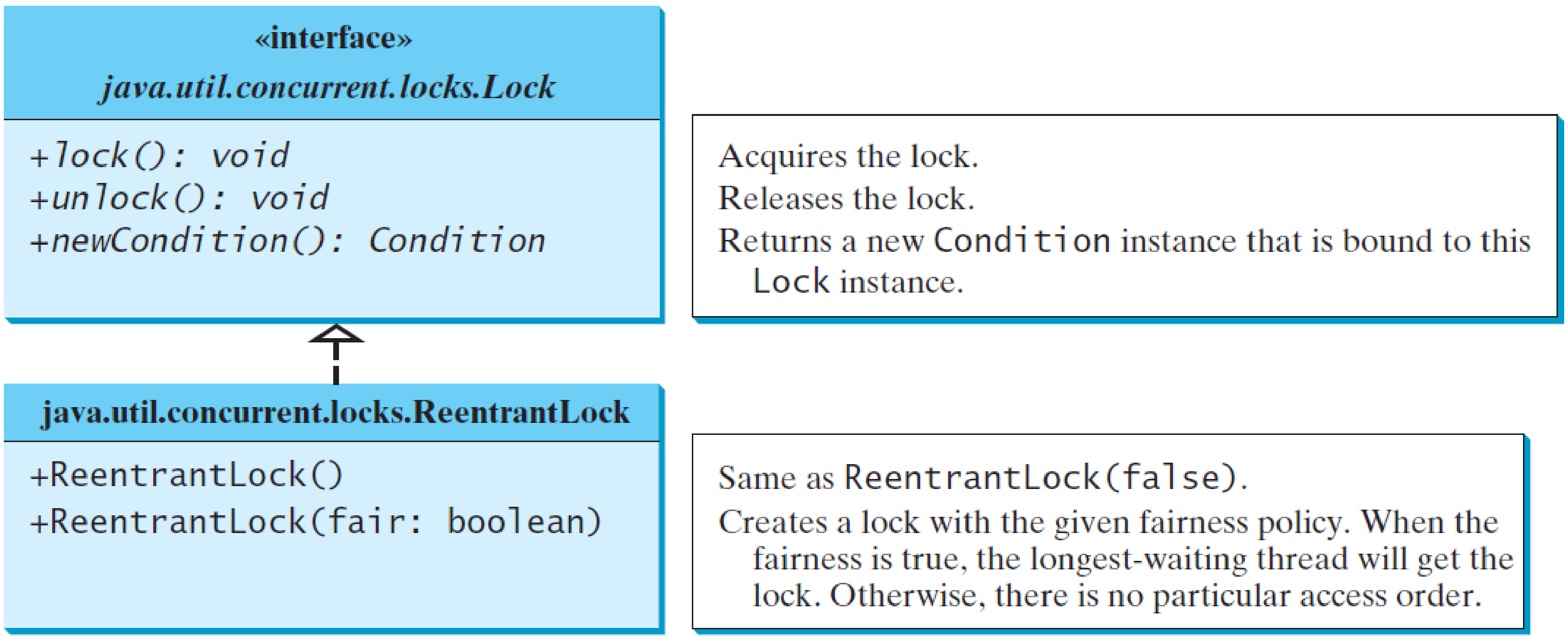
  → This is the responsibility of owner of lock

  → However, it IS the owner

  → Therefore, it should unlock the lock itself

  → It cannot do this, because it is blocked waiting for the lock to be unlocked.

The Lock Interface



ReentrantLocks and How to Use Them

```
public class X {
    private final ReentrantLock lock = new ReentrantLock();

    public void m() {
        lock.lock();
        try {
            // critical section
        } finally {
            lock.unlock()
        }
    }
}
```

Proper Sequencing

- So far we have seen how to achieve mutually exclusive access to shared data by guarding the critical sections.
- However, we also care about proper sequencing of threads when dependencies are present.
- For example, if thread A produces data and B consumes it, then B has to wait until A has produced something before starting to consume.

Busy-waiting / Spinning

- A naive solution is to repeatedly check to see if some predicate is true:

```
int amount_to_withdraw = Util.randomInRange(50, 150);
while (account.getBalance() < amount_to_withdraw)  {
        // do nothing
}
```

- Very wasteful in terms of CPU cycles!
- Still has check-then-act issues.
- Use a wait and signal mechanism instead: Conditions.

Conditions

- Essentially a queue of threads waiting to be signaled.
- Associated with a single lock that must be held when testing the predicate.
- Has an **await()** and **signalAll()** method.
- Does NOT check whether predicate holds.
- Is only there as the signaling mechanism.

Conditions (2)

In one method:

- **Acquire** associated **lock**.
- Check **predicate**.
- If not satisfied, call **await** on Condition object until a **signal** has been received.
- Acquire lock and check predicate **again**!
- Do your work.
- **Release** the **lock**.

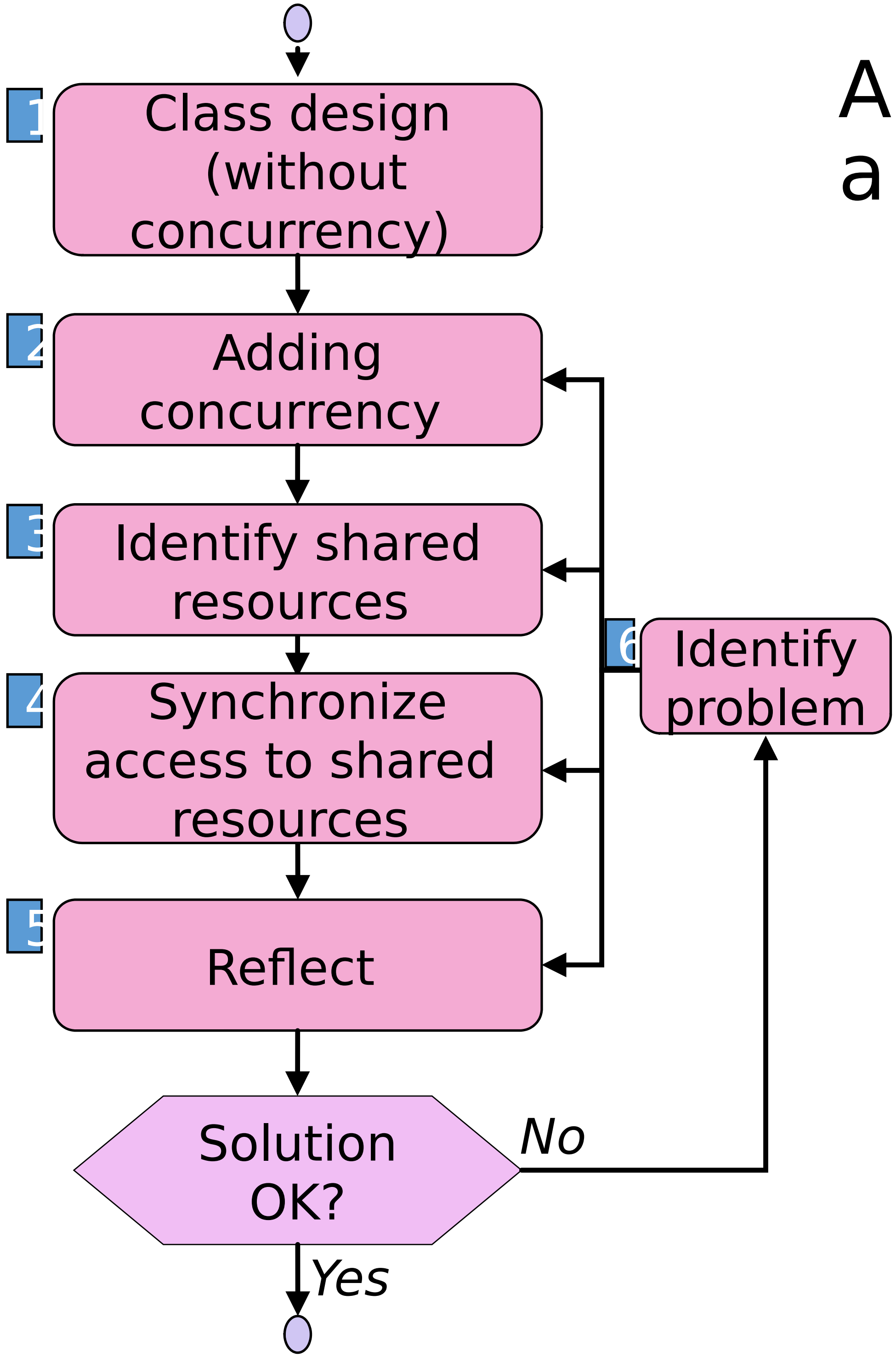
Conditions (3)

In second method:

- **Acquire** associated **lock**.
- Do your work that might lead predicate to be evaluated to true.
- Send **signalAll** through Condition object.
- **Release** the **lock**.

What Is Wrong Here?

```
int amount_to_withdraw = Util.randomInRange(50, 150);
if (account.getBalance() < amount_to_withdraw)  {
      newDeposit.await();
}
// Withdraw the money
```



Approach for designing a concurrent program

- Steps 1 and 2 involve creating a concurrent program;
- Steps 3 and 4 are necessary to solve the potential problems that are introduced by using threads;
- Step 5 is a final check

Demo: Bounded Producer/Consumer

Assignment: Taxi