Course of Programming in Java

by Łukasz Stafiniak

Email: lukstafi@gmail.com, lukstafi@ii.uni.wroc.pl
Web: www.ii.uni.wroc.pl/~lukstafi

Thinking in Java

Chapter 22. Concurrency

The Java Programming Language

Chapter 14. Threads

Concurrency

- **Concurrency** means organizing a program into threads that are / appear to be simultaneous *threads* of computation.
- **Parallelism** is employing multiple processors to speed up computation.
- **Distributed computing** deals with parallelism across multiple machines.
- **Multithreading** is a form of concurrency with threads operating on shared objects.
- **Cooperative multithreading** works without help of the system, like in our *Settlers* simulation. Pieces of code in a single method are *atomic*.
- In Java, threads are based on system *pthreads* (POSIX threads).
- A race hazard exists when two threads can potentially modify the same piece of data in an interleaved way that can corrupt data.
  - In a bank example, imagine that two costumers walk up to two bank tellers to deposit money into the same account. Each teller goes to the filing cabinet to get the current account balance and gets the same information. Then...

Applications of Concurrency

- Simulation.
- User interfaces.
- (In particular) Interactive content. Reactive systems.
- Input / Output. (Especially over internet.)
- Web servers.
- Speeding-up computation (parallelization).
- ...

Creating Threads

```
public class PingPong extends Thread {
    private String word;  // what word to print
    private int delay;    // how long to pause
    public PingPong(String whatToSay, int delayTime) {
        word = whatToSay;
        delay = delayTime;
    }
    public void run() {
        try {
            for (;;) {
                System.out.print(word + " ");
                Thread.sleep(delay); // wait until next time
            }
        } catch (InterruptedException e) {
            return;              // end this thread
        }
    }
    public static void main(String[] args) {
        new PingPong("ping",  33).start(); // 1/30 second
        new PingPong("PONG", 100).start(); // 1/10 second
    }
}
```

- First configure a task (its priority, whether it is a daemon).
- Then invoke `start`, only once for each thread (invoking it again results in an `IllegalThreadStateException`).
- The standard implementation of `Thread.run` does nothing. To get a thread that does something you must either extend `Thread` to provide a new `run` method,
- or create a `Runnable` object and pass it to the thread's constructor. This lets you extend a base class not extending `Thread`.
- You can obtain the `Thread` object for the currently running thread by invoking the static method `Thread.currentThread`.
- Starting a thread in a constructor is handy but can be risky if a class can be extended.
- Thread objects are only garbage-collected when they stop running.
  - *Daemon* threads are terminated when remaining threads end.
- `java.util.concurrent` provides some higher-level approaches to arrange concurrent computation.

Using an inner `Runnable` class, `run()` is not exposed outside of its thread.

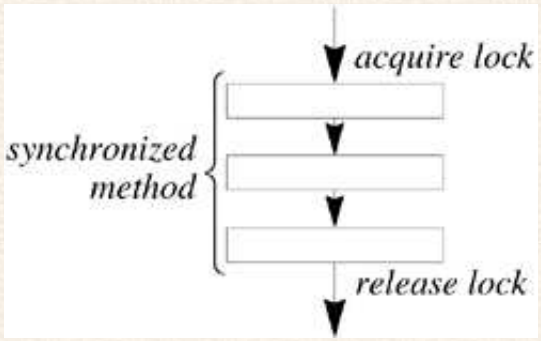
```
class PrintServer2 {
    private final PrintQueue requests = new PrintQueue();
    public PrintServer2() {
        Runnable service = new Runnable() {
            public void run() {
                for(;;)
                    realPrint(requests.remove());
            }
        };
        new Thread(service).start();
    }
    public void print(PrintJob job) {
        requests.add(job);
    }
    private void realPrint(PrintJob job) {
        // do the real work of printing
    }
}
```

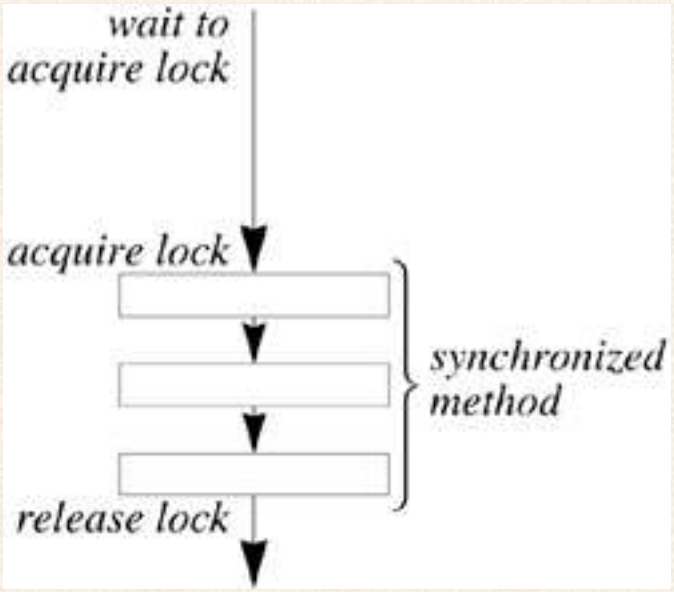
Thread Scheduling

- Do not rely on thread priority for algorithm correctness. (It is only to tune efficiency.)
  - If you use priorities at all, the continuously running part of your application should run in a lower-priority thread than the thread dealing with rarer events such as user input.
  - A thread that does continual updates is often set to `NORM_PRIORITY-1` so that it doesn't hog all available cycles, while a user interface thread is often set to `NORM_PRIORITY+1`.
- Assume that a thread could be *pre-empted* at any time, and so you need to always protect access to shared resources.
- `sleep(long delay) throws InterruptedException` puts the currently executing thread to sleep for at least the specified time.
- `yield()` provides a hint to the scheduler that the current thread need not run at the present time.

Synchronization

- **Critical sections** or **critical regions**: objects / resources that are suspect to interference of operations by different threads.
- Acquiring a lock.
  - In the bank, tellers synchronize their actions by putting notes in the files and agreeing to the protocol that a note in the file means that the file can't be used.
- `synchronized` methods





- Locks are owned per thread, so invoking a synchronized method from within another method synchronized on the same object will proceed without blocking, releasing the lock only when the outermost synchronized method returns.
- The lock is released as soon as the synchronized method terminates.
- Use getter and setter methods with private fields to synchronize on them.

```
public class BankAccount {
    private long number;    // account number
    private long balance;   // current balance (in cents)
    public BankAccount(long initialDeposit) {
        balance = initialDeposit;
    }
    public synchronized long getBalance() {
        return balance;
    }
    public synchronized void deposit(long amount) {
        balance += amount;
    }
}
```

- A static `synchronized` method acquires the lock of the `Class` object for its class. Two threads cannot execute static synchronized methods of the same class at the same time, just as two threads cannot execute synchronized methods on the same object at the same time.
- The `synchronized` statement enables you to execute synchronized code that acquires the lock of any object, not just the current object, or for durations less than the entire invocation of a method.

```
/** make all elements in the array non-negative */
public static void abs(int[] values) {
    synchronized (values) {
        for (int i = 0; i < values.length; i++) {
            if (values[i] < 0)
                values[i] = -values[i];
        }
    }
}
```

Benefits of the `synchronized` statement

- Synchronization affects performance – a general rule is to hold locks for as short a period as possible.
- Different groups of methods within a class can act on different data within that class. Instead of making all the methods synchronized, define separate objects to be used as locks for each such group.

```
class SeparateGroups {
    private double aVal = 0.0;
    private double bVal = 1.1;
    protected final Object lockA =
        new Object();
    protected final Object lockB =
        new Object();
    public double getA() {
        synchronized (lockA) {
            return aVal;
        }
    }
    public void setA(double val) {
        synchronized (lockA) {
            aVal = val;
        }
    }
    public double getB() {
        synchronized (lockB) {
            return bVal;
        }
    }
    public void setB(double val) {
        synchronized (lockB) {
            bVal = val;
        }
    }
    public void reset() {
        synchronized (lockA) {
            synchronized (lockB) {
                aVal = bVal = 0.0;
            }
        }
    }
}
```

- Synchronize an inner object with the enclosing object.

```
public class Outer {
    private int data;
    // ...
    private class Inner {
        void setOuterData() {
            synchronized (Outer.this) {
                data = 12;
            }
        }
    }
}
```

- If you need to protect access to static data from within non-static code (or synchronize with static methods):

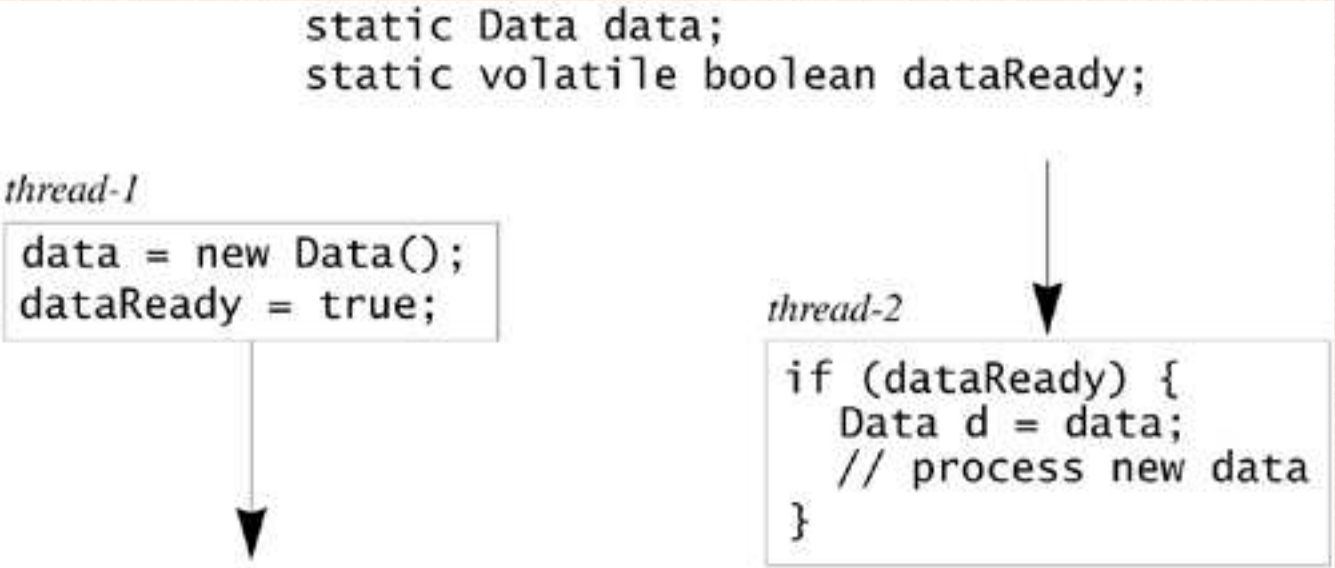
```
Body() {
    synchronized (Body.class) {
        idNum = nextID++;
    }
}
```

  - We could also define a `static synchronized` method `getNextID`.

- When you need to synchronize operations on multiple objects, the synchronize statement is most practical.
- Synchronized methods are more "object-oriented" – the object protects itself (better encapsulation, not relying on the caller).
- `synchronized` statement approach is called *client-side synchronization* and `synchronized` methods approach is *server-side synchronization*.
- `java.util.concurrent.locks` has explicit `Lock` objects (implemented by `ReentrantLock`) with `lock()` and `unlock()` methods, for flexible control over synchronization.
  - E.g. acquiring lock for the next object right before releasing lock for a given object.

Synchronizing fields

- Without synchronization on access, there are no guarantees on what another thread reads (can read an old value due to cache effects)
- `volatile` fields synchronize between writes and reads (but do not provide atomicity for more complex operations).
- "Life events" of threads (creating, ending) synchronize with those who observe them (e.g. after `join()` the results of the ended thread are synchronized).



Synchronizing Data Structures

- Just as reading and writing of fields, getting and putting data into data structures needs to be synchronized.
- Standard collections are not synchronized, for efficiency.
- `Collections.synchronizedCollection` and related wrappers (e.g. `Collections.synchronizedSet`) return variants of collections with synchronized (therefore "atomic") operations.
- Now you can synchronize on the collection to prevent interference

```
Map m = Collections.synchronizedMap(new HashMap());
// ...
synchronized (m) { if (!m.containsKey(key)) m.put(key, value); }
```

  - Even if outside code does not use `synchronized (m)`, it will not interfere with the above block.
- Iterators must be used inside synchronized blocks.
- If you need synchronized maps or queues, consider *lock-free containers* `ConcurrentHashMap` and `ConcurrentLinkedQueue` from `java.util.concurrent`.

`ThreadLocal` Variables

- `ThreadLocal<T>` is a variable that has independent values in each separate thread – like a new field in each thread class, but without the need to actually change any thread class.
- Accessor methods `get()` and `set()`, initialized by `initialValue()` (default: `null`).
- Local variables inside `run()` are already thread-local, but `ThreadLocal` provides *global* thread-local variables.
  - `ThreadLocal` also allows a shared structure to have unshared (i.e. thread-local) parts.
- Caution: with thread-pooling, running a new task does not re-initialize `ThreadLocal` variables.

```
public class Operations {
    static Operations userOps = new Operations(); // global thread-local variable
    private static ThreadLocal<User> users =
        new ThreadLocal<User>() {
            /** Initially start as the "unknown user". */
            protected User initialValue() {
                return User.UNKNOWN_USER;
            }
        };
    public static void setUser(User newUser) {
        validate(newUser); users.set(newUser);
    }
    public void setValue(int newValue) {
        User user = currentUser();
        if (!canChange(user)) throw new SecurityException();
        // ... modify the value ...
    } // ...
}
public class Console implements Runnable {
    User user;
    public void run () {
        Operations.userOps.setUser (user);
        while (!Thread.currentThread().isInterrupted()) {
            // ...
        }
    } // ...
}
```

Deadlocks

```
class Friendly {
    private Friendly partner;
    private String name;
    public Friendly(String name) { this.name = name; }
    public synchronized void hug() {
        System.out.println(Thread.currentThread().getName()+
              " in " + name + ".hug() trying to invoke " +
              partner.name + ".hugBack()");
        partner.hugBack();
    }
    private synchronized void hugBack() {
        System.out.println(Thread.currentThread().getName()+
              " in " + name + ".hugBack()");
    }
    public void becomeFriend(Friendly partner) { this.partner = partner; }
}
...
    final Friendly jareth = new Friendly("jareth");
    final Friendly cory = new Friendly("cory");
    jareth.becomeFriend(cory); cory.becomeFriend(jareth);
    new Thread(new Runnable() {
        public void run() { jareth.hug(); } }, "Thread1").start();
    new Thread(new Runnable() {
        public void run() { cory.hug(); } }, "Thread2").start();
```

- The following scenario is possible:
  1. Thread number 1 invokes `synchronized` method `jareth.hug`. Thread number 1 now has the lock on `jareth`.
  2. Thread number 2 invokes `synchronized` method `cory.hug`. Thread number 2 now has the lock on `cory`.
  3. Now `jareth.hug` invokes `synchronized` method `cory.hugBack`. THRead number 1 is now blocked waiting for the lock on `cory` (currently held by thread number 2) to become available.
  4. Finally, `cory.hug` invokes `synchronized` method `jareth.hugBack`. Thread number 2 is now blocked waiting for the lock on `jareth` (currently held by thread number 1) to become available.
- One common technique to avoid deadlocks is resource ordering: assign an order on all locks and always acquire them in that order.
  - E.g. once one thread has the first lock, the second thread will block trying to acquire that lock, and then the first thread can safely acquire the second lock.

Interrupting work

- `interrupt()` sets the `isInterrupted()` flag – `interrupted()` checks and resets the flag.
- Check proactively for interruption, at points where partial results can be prepared, or where stopping and cleaning-up is most reasonable.
- `InterruptedException` raised by `sleep()`, `wait()`, `join()` when `interrupt()` called on a thread and when it already `isInterrupted`.
  - remember to handle interruption in your methods "manually", e.g. by raising `InterruptedException`

```
void tick(int count, long pauseTime) {
    try {
        for (int i = 0; i < count; i++) {
            System.out.println('.'); System.out.flush();
            Thread.sleep(pauseTime);
        }
    } catch (InterruptedException e) {
        Thread.currentThread().interrupt();
    }
}
```

```
double output = 0.0;
public void run () {
    double sign = 1.0;
    for (double i=0.0; i < count && !Thread.interrupted(); i += 1.0) {
        output += sign / (2.0*i + 1.0);
        sign *= -1.0;
    }
    output *= 4.0;
}
```

- Unfortunately, `interrupt()` does not raise `InterruptedException` in threads blocked at entrance to `synchronized` method or statement, therefore cannot be used to break out of deadlocks.
- `interrupt()` does not interrupt standard IO as well.
  - One way to force interrupt is to `close()` IO streams from outside the thread.
  - Another is to use the `java.nio` (Non-blocking Input-Output) package, which handles interrupts.
- Recall `ReentrantLock` with `lock()` and `unlock()` methods – blocking using its method `lockInterruptibly()` does handle interrupts.

Waiting for a Thread to Complete

```
class CalcThread extends Thread {
    private double result;
    public void run() { result = calculate(); }
    public double getResult() { return result; }
    public double calculate() {
        // ... calculate a value for "result"
    }
}
class ShowJoin {
    public static void main(String[] args) {
        CalcThread calc = new CalcThread();
        calc.start();
        doSomethingElse();
        try {
            calc.join(); // guarantees that calc.run() finished
            System.out.println("result is " + calc.getResult());
        } catch (InterruptedException e) {
            System.out.println("No answer: interrupted");
        }
    }    // ... definition of doSomethingElse ...
}
```

Threads as Functions

- `Callable<T>` can be used in place of `Runnable` to compute a value.
- `call()` (instead of `run()`) returns the value of type `T`.
- Must be invoked using an `ExecutorService` `submit()` (instead of its own `start()` or of `ExecutorService` `execute()`)
- that produces a `Future<T>` object, which can be checked with `isDone()`, and the result extracted with `get()` (which blocks until the computation finishes).
- For example, while waiting to `get()` one of several results, all of them are being computed in parallel.

```
import java.net.*;
import java.io.*;
import java.util.concurrent.*;
import java.util.*;
class ReadWebPage implements Callable<String> {
    private String address; private int numLines; private String lineBreak;
    public ReadWebPage (String address, int numLines, String lineBreak) {
      this.address = address; this.numLines = numLines; this.lineBreak = lineBreak;
    }
    public String call () {
        try {
            Scanner in = new Scanner(new URL(address).openStream());
            int i = numLines;
            StringBuffer result = new StringBuffer ();
            while (in.hasNextLine() && --i >= 0) {
                String line = in.nextLine();
                result.append (line + lineBreak);
            }
            in.close();
            return result.toString ();
        } catch (MalformedURLException me) {
            System.err.println(me); return "ERROR: Malformed URL "+address;
        } catch (IOException ioe) {
            System.err.println(ioe); return "ERROR: IO error reading "+address;
        }
    }
}
```

```
public class ReadWebPages {
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        ArrayList<Future<String>> results =
            new ArrayList<Future<String>>();
        for(String address : args) {
            System.out.println ("Reading "+address+"...");
            results.add(exec.submit(new ReadWebPage(address, 20, "\n")));
        }
        for(Future<String> fs : results)
            try {
                System.out.println ("\n============================");
                // get() blocks until completion:
                System.out.println(fs.get());
            } catch(InterruptedException e) {
                System.out.println(e);
                return;
            } catch(ExecutionException e) {
                System.out.println(e);
            } finally {
                exec.shutdown();
            }
    }
}
```

Executors

- `ExecutorServices` manage groups of threads.
- `shutdownNow()` interrupts all tasks in a group.
- To interrupt a single task, `submit()` it instead of `execute()` to be able to interact with it using the returned `Future<?> f` – the task can be interrupted by `f.cancel(true)` (does not support partial results).
- `ScheduledThreadPoolExecutor` can schedule a task at a given time in the future using `schedule()`, or to be run periodically (i.e. repeated at a regular interval) using `scheduleAtFixedRate()`.

Cooperation Between Tasks

- `wait()` stops a thread synchronizing on an object until some other thread calls `notifyAll` on this object. Using `notify` is an optimization that can be applied only when:
  - All threads are waiting for the same condition
  - At most one thread can benefit from the condition being met
  - This is true for all possible subclasses

```
class PrintQueue {
    private SingleLinkQueue<PrintJob> queue = new SingleLinkQueue<PrintJob>();
    public synchronized void add(PrintJob j) {
        queue.add(j);
        notifyAll();     // Tell waiters: print job added
    }
    public synchronized PrintJob remove()
        throws InterruptedException  // wait() can throw it
    {
        while (queue.size() == 0)
            wait();      // Wait for a print job
        return queue.remove();
    }
}
```

- Wait in a loop checking a condition since there might be spurious wake-ups, and more importantly the condition can change in the meanwhile
- The lock on the object is released during `wait()` – e.g. `synchronized` methods can be called while some of them are waiting. The thread will only resume when it can acquire the lock again (so, can be blocked).
  - This **is not the case** for `sleep()` and `yield()` – they keep the lock.
- `public final void wait(long timeout)` time-outs waiting if not woken before the specified number of milliseconds;
  - but will only resume once it acquires the lock.
- `notifyAll()` notifies all threads of an object, `notify()` notifies only one thread.
- `wait` can be only called from synchronized code.
  - `IllegalMonitorStateException` if you attempt to invoke these methods on an object when you don't hold its lock.
- Car Waxing example from *"Thinking in Java"*: See file `WaxOMatic.java`.

Blocking Queues

- A *synchronized queue* (a `BlockingQueue`) only allows one task at a time to insert or remove an element: synchronizes on its operations.
- A thread trying to `take()` from an empty `BlockingQueue` (or `PriorityBlockingQueue`) is suspended, and resumes when the queue becomes non-empty.
  - `ArrayBlockingQueue` has a bounded capacity, and a thread trying to `put(e)` when the queue is full, is suspended until queue has space.
- No need to use `wait()` and `notifyAll()`. See file `ToastOMatic.java` from *"Thinking in Java"*.
- Queues between threads can also serve as *message queues*: instead of calling synchronized methods of an object, we can write a thread that reads messages from a queue and dispatches corresponding actions, and later send messages to the object (*active object*, or *agent*).

`CyclicBarrier` **from** `java.util.concurrent`

- `CyclicBarrier` is used to periodically synchronize several tasks (together at once).
- A task tells that it is ready by calling `await()` on a `CyclicBarrier`.
- When all tasks are ready, `CyclicBarrier`'s `run()` is called, and then the tasks resume (and can call `await()` again).
- See file `HorseRace.java` from *"Thinking in Java"*.

Examples

- Queues example from *"Thinking in Java"*: See file `ToastOMatic.java`.
- Workload example from *"Thinking in Java"*: See file `BankTellerSimulation.java`.