CSC 7203

J <u>Paul</u> Gibson, D311

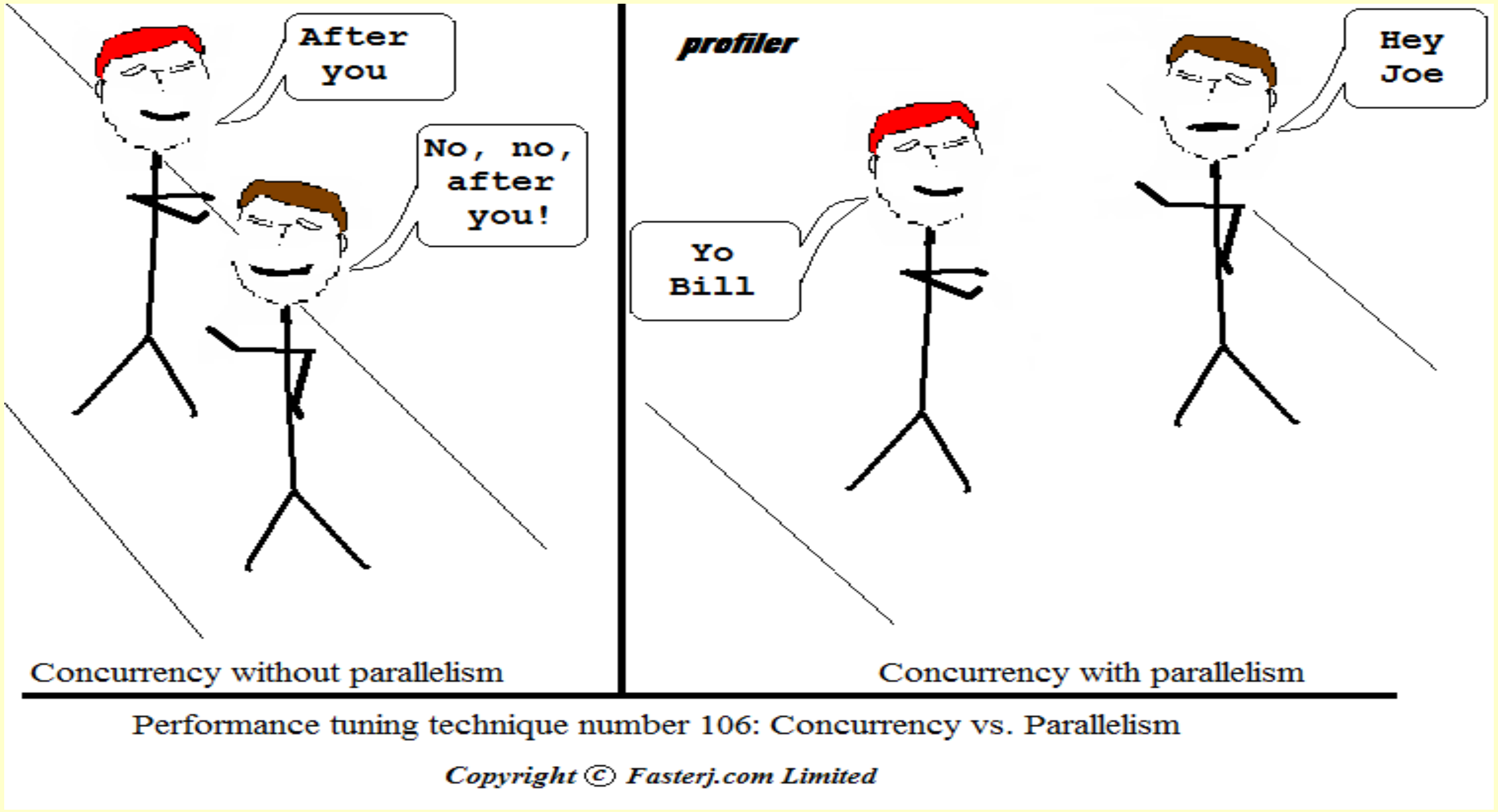
paul.gibson@telecom-sudparis.eu

http://www-public.it-sudparis.eu/~gibson/Teaching/CSC7203/

Threads(in Java)

…/~gibson/Teaching/CSC7203/CSC7203-AdvancedOO-L7-Threads.pdf

Processes and Threads


After you
No, no, after you!
profiler
Yo Bill
Hey Joe
Concurrency without parallelism
Concurrency with parallelism
Performance tuning technique number 106: Concurrency vs. Parallelism
Copyright © Fasterj.com Limited

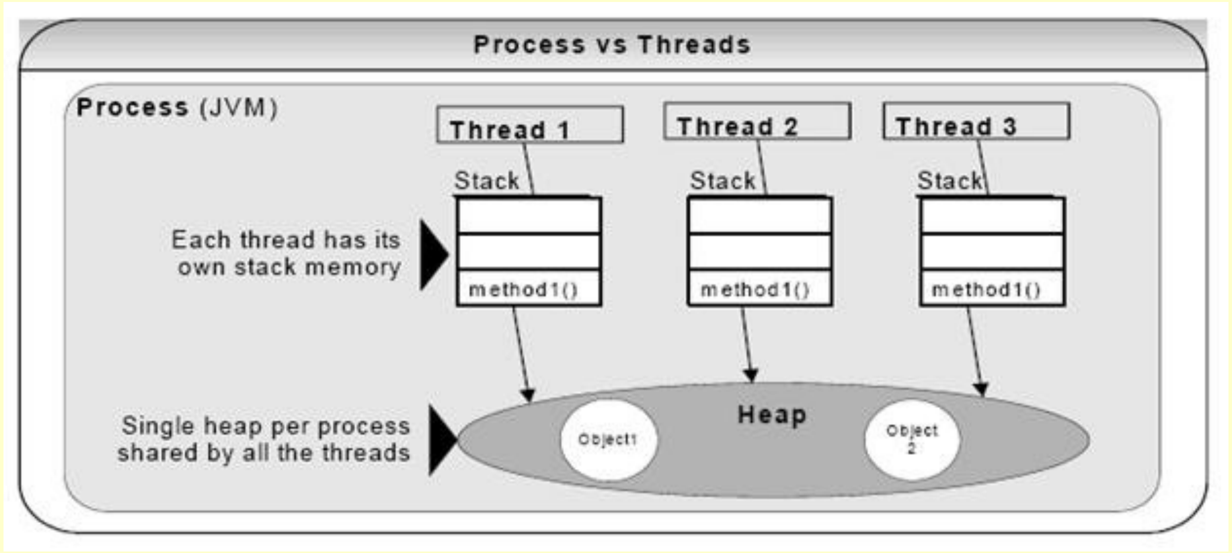
Processes and **Threads** are the two fundamental units of execution in a concurrent program.

Processes and Threads

•In Java, concurrent programming is mostly thread-based.

•Processing time for each core in a system is shared among processes and threads through an OS feature called time slicing.

•Concurrency is possible even on simple systems, without multiple processors or execution cores.



http://www.java-forums.org/blogs/thread/

Processes

Self-contained execution environment.

Independent set of basic run-time resources, such as memory space.

A single application may be implemented by a set of cooperating processes.

Most operating systems support *Inter Process Communication* (IPC) resources.

IPC can also used for communication between processes on different systems.

Most implementations of the JVM run as a single process.

Threads

Also known as *lightweight processes*.

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 has advantages and disadvantages … can you think of them?

Multithreaded execution is essential in Java:

- every application has at least one thread
- "system" threads that do memory management, event/signal handling, etc.

In programming, we start with just one thread, called the *main thread*.

Any thread (including the main thread) can create new threads.

Threads in Java: some additional reading

Fixing The Java Memory Model, William Pugh, 1999.

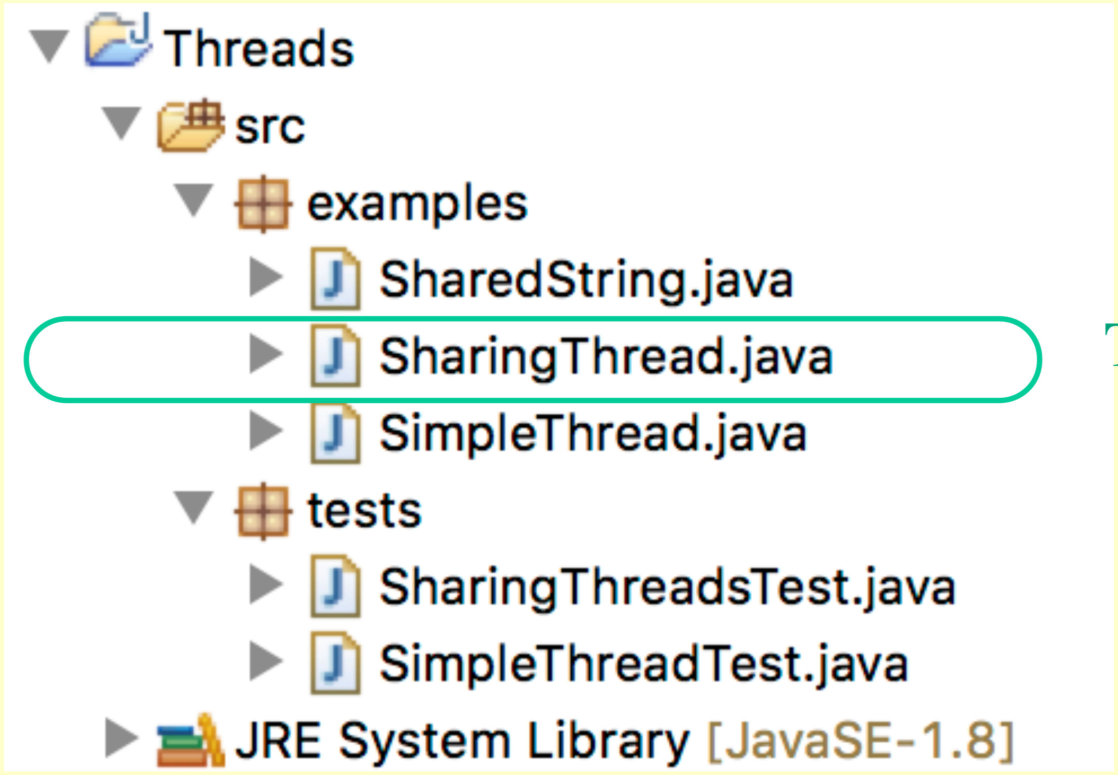
The Problem with Threads, Edward Lee, 2006.

Java Thread Programming, by Paul Hyde ISBN: 0672315858
Sams 1999

Concurrent Programming in Java™: Design, Principles and Patterns, Second Edition, By Doug Lea, ISBN: 0-201-31009-0
Addison Wesley, 2001

Thread Example

Download the code Threads.zip from the web site and import it into Eclipse



```
▼ Threads
  ▼ src
    ▼ examples
      ▶ SharedString.java
      ▶ SharingThread.java
      ▶ SimpleThread.java
    ▼ tests
      ▶ SharingThreadsTest.java
      ▶ SimpleThreadTest.java
  ▶ JRE System Library [JavaSE-1.8]
```

TO DO

Thread Example

```
public class ThreadExample {

    public static void main (String[] args) {
        System.out.println("Starting Thread main");
        new SimpleThread("Add1", '1').start();
        new SimpleThread("Add2", '2').start();
        System.out.println("Finishing Thread main");
    }
}
```

Thread Example - typical output

```
Starting Thread main
Finishing Thread main
String Add2 extended to 2
String Add2 extended to 22
String Add2 extended to 222
String Add1 extended to 1
String Add1 extended to 11
String Add2 extended to 2222
String Add2 extended to 22222
No more increments left for threadAdd2
String Add1 extended to 111
String Add1 extended to 1111
String Add1 extended to 11111
No more increments left for threadAdd1
```

Thread Example - SimpleThread Code

```
/* see -
http://docs.oracle.com/javase/6/docs/api/java/lang/Thread.html
*/

class SimpleThread extends Thread {
String stringofchars;
char increment;

    public SimpleThread(String str,  char inc) {
        super(str);
        stringofchars = "";
        increment = inc;
    }
    public void run() {
        for (int i = 0; i < 5; i++) {

            try {
                sleep((int)(Math.random() * 3000));
            } catch (InterruptedException e) {}
        stringofchars = stringofchars + increment;
        System.out.println("String " + getName()+
                              " extended to "+ stringofchars );
        }
        System.out.println("No more increments left for
                         thread" + getName());
    }
}
```

`IllegalThreadStateException`

The runtime system throws an `IllegalThreadStateException` when you call a method on a thread and that thread's state does not allow for that method call. (See the state machine diagram in later slides)

So, when you call a thread method that can throw an exception, you must either catch and handle the exception, or specify that the calling method throws the uncaught exception.

The sleep method can also throw an `InterruptedException`, and so we needed a try/catch in the previous code:

```
try {
        sleep((int)(Math.random() * 3000));
      } catch (InterruptedException e) {}
```

Sharing Thread Problem

The previous example showed how two independent threads execute concurrently.

Threads can also share data/objects and so their concurrent behaviours are inter-dependent.

We wish to change the previous code so that the 2 threads update the same string of characters.

We will do this using a SharedString class

Sharing Thread Problem

Your task is to code the `class SharingThread`

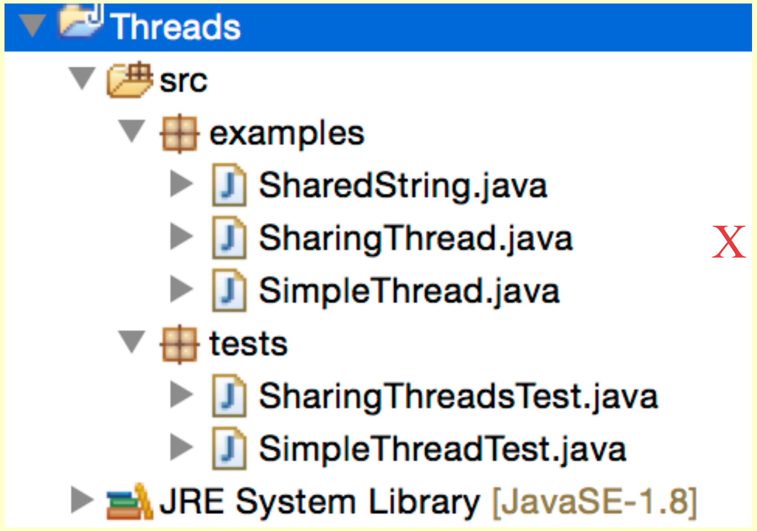
```
 class SharedString {

public SharedString(){str ="";}

public String str;

public  void add (char c){str = str + c;}

public String toString () {return str;}
}
```



```
Threads
  src
    examples
      SharedString.java
      SharingThread.java
      SimpleThread.java
    tests
      SharingThreadsTest.java
      SimpleThreadTest.java
  JRE System Library [JavaSE-1.8]
```

X

Threads.zip

```
public class SharingThreadsTest {

    public static void main (String[] args) {

    SharedString soc = new SharedString();
        new SharingThread("SharingAdda", soc, 'a').start();
        new SharingThread("SharingAddb", soc, 'b').start();
    }

}
```

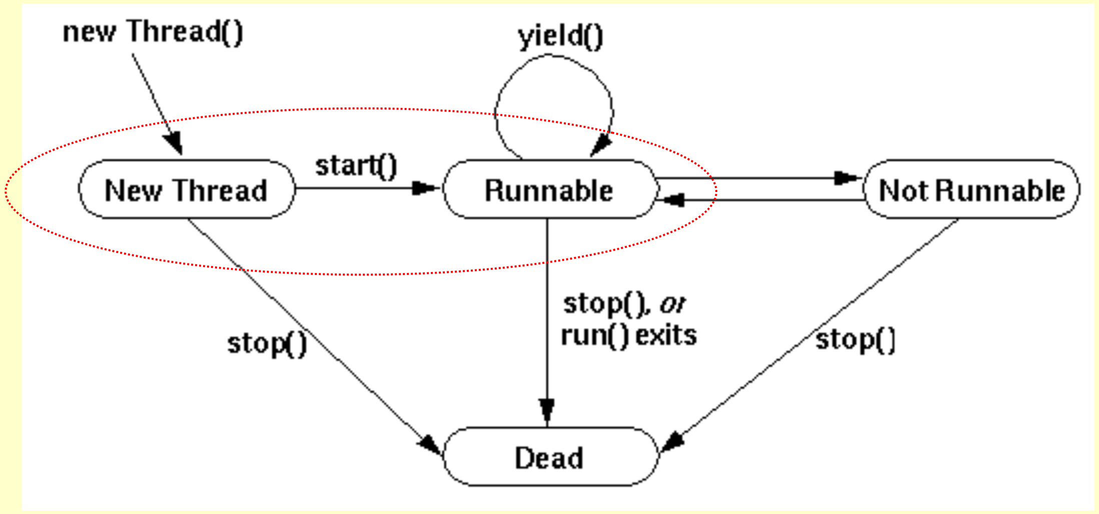
Sharing Thread Problem

We want the output from this code to produce, typically:

```
Shared String extended by SharingAddb to b
Shared String extended by SharingAddb to bb
Shared String extended by SharingAdda to bba
Shared String extended by SharingAddb to bbab
Shared String extended by SharingAddb to bbabb
Shared String extended by SharingAdda to bbabba
Shared String extended by SharingAddb to bbabbab
No more increments left SharingAddb
Shared String extended by SharingAdda to bbabbaba
Shared String extended by SharingAdda to bbabbabaa
Shared String extended by SharingAdda to bbabbabaaa
No more increments left SharingAdda
```

TO DO: Your task is to code the **`class`** `SharingThread` **`extends`** `Thread {}` to provide this behaviour

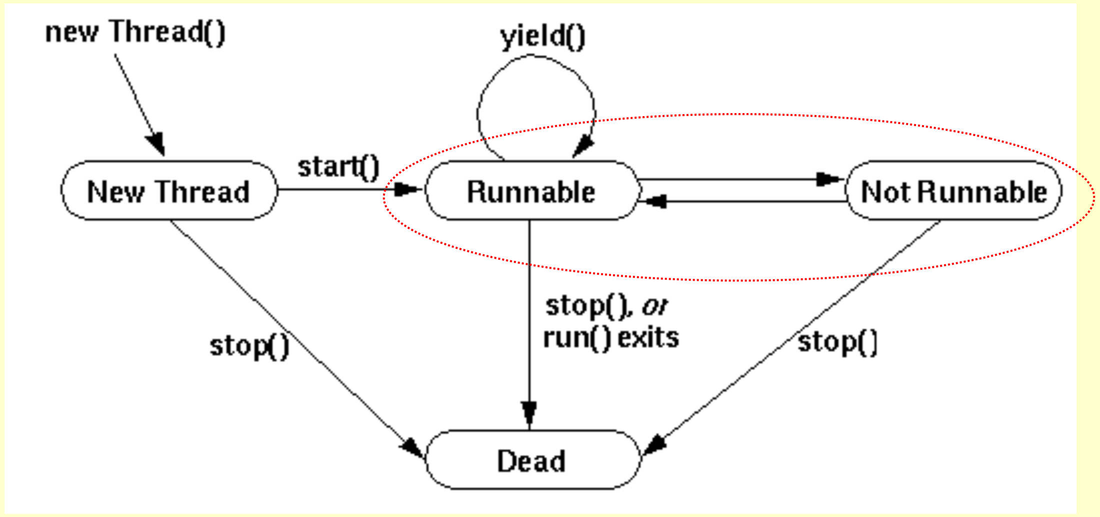
Thread State Machine: an abstraction of the complete diagram



The `start()` method creates the system resources necessary to run the thread, schedules the thread to run, and calls the thread's `run()` method.

The next state state is "`Runnable`" rather than "`Running`" because the thread might not actually be running when it is in this state.

Thread State Machine: an abstraction of the complete diagram



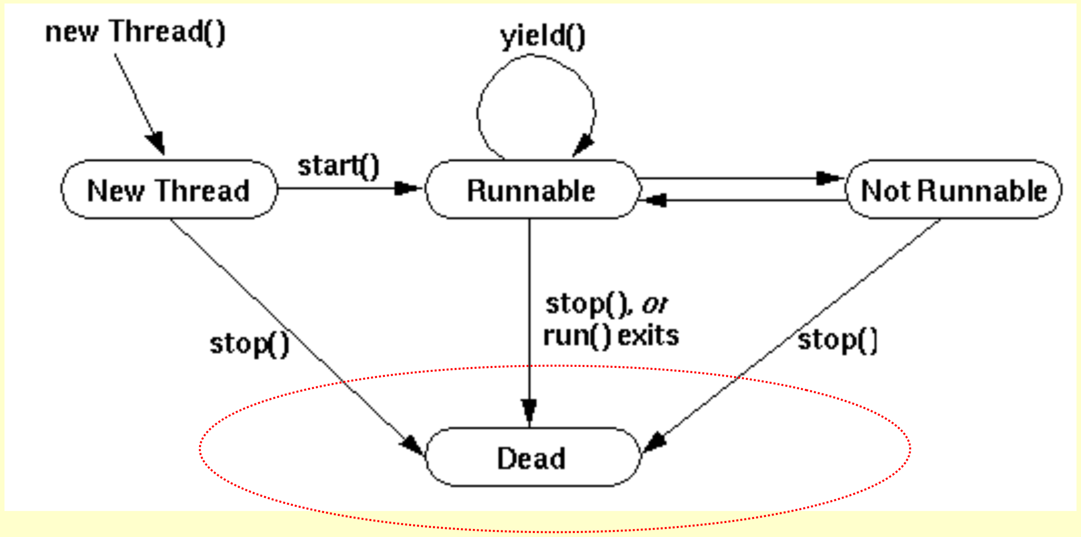
A thread enters the "`Not Runnable`" state when:

- `sleep()` is called.
- `suspend()` is called.
- The thread uses its `wait()` method to wait on a condition variable.
- The thread is blocking on I/O.

A thread leaves the "`Not Runnable`" state when <u>a matching condition</u> is met:

- `sleep()` is completed.
- `resume()` is called
- object owning the variable calls `notify()` or `notifyAll()`
- I/O completes

Thread State Machine: an abstraction of the complete diagram



A thread dies naturally when its `run()` method exits normally

You can also kill a thread at any time by calling its `stop()` method

QUESTION: What should happen if an exception occurs inside a thread?

Threads and Synchronization Issues

Threads can share state (objects)

This is very powerful, and makes for very efficient inter-thread communication

However, it makes two kinds of errors possible:

- *thread interference*, and
- *memory inconsistency*.

Java provides a *synchronization "tool"* in order to avoid these types of errors.

Thread Interference

Interference happens when two operations, running in different threads, but acting on the same data, *interleave*. This means that the two operations consist of multiple steps, and the sequences of steps overlap. Because they are unpredictable, thread interference bugs can be difficult to detect and fix.

Consider a simple class called Counter

```
    class Counter {
        private int c = 0;
                public void increment() {c++;}

        /*                Multiple steps of c++
1.                       Retrieve the current value of c.
2.                      Increment the retrieved value by 1.
3.                     Store the incremented value back in c.
        */

        public void decrement() {c--;}
        public int value() {return c;}
    }
```

If a Counter object is referenced from multiple threads, interference between threads may give rise to unexpected behaviour.

Memory inconsistency

Consider the following example.

```
        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"`.

But, in this example, the value printed out might well be "0", because there's no guarantee that thread A's change to counter will be visible to thread B — unless the programmer has established a ***happens-before*** relationship between these two statements.

There are several actions that create ***happens-before*** relationships.

The simplest technique/tool is to use ***`synchronization`***

Synchronized methods, example:

```
public class SynchronizedCounter {
    private int c = 0;
    public synchronized void increment() {c++;}
    public synchronized void decrement() {c--;}
    public synchronized int value() {return c;}
}
```

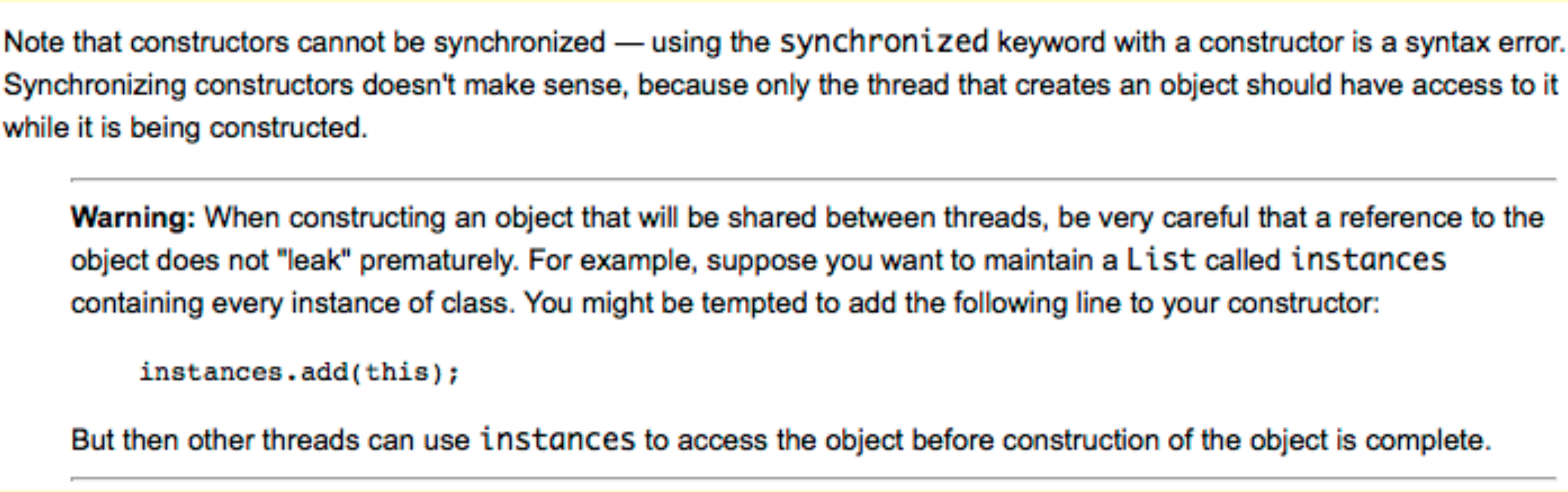
Two invocations of synchronized methods on the same object cannot interleave. When one thread is executing a synchronized method for an object, all other threads that invoke synchronized methods for the same object block (suspend execution) until the first thread is done with the object.

When a synchronized method exits, it automatically establishes a *happens-before* relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes to the state of the object are visible to all threads.

Synchronization is effective for keeping systems *safe*, but can present problems with *liveness*

Java Constructors cannot be synchronized

http://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html



Note that constructors cannot be synchronized — using the `synchronized` keyword with a constructor is a syntax error. Synchronizing constructors doesn't make sense, because only the thread that creates an object should have access to it while it is being constructed.

Warning: 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.