Dr.-Ing. Michael Eichberg

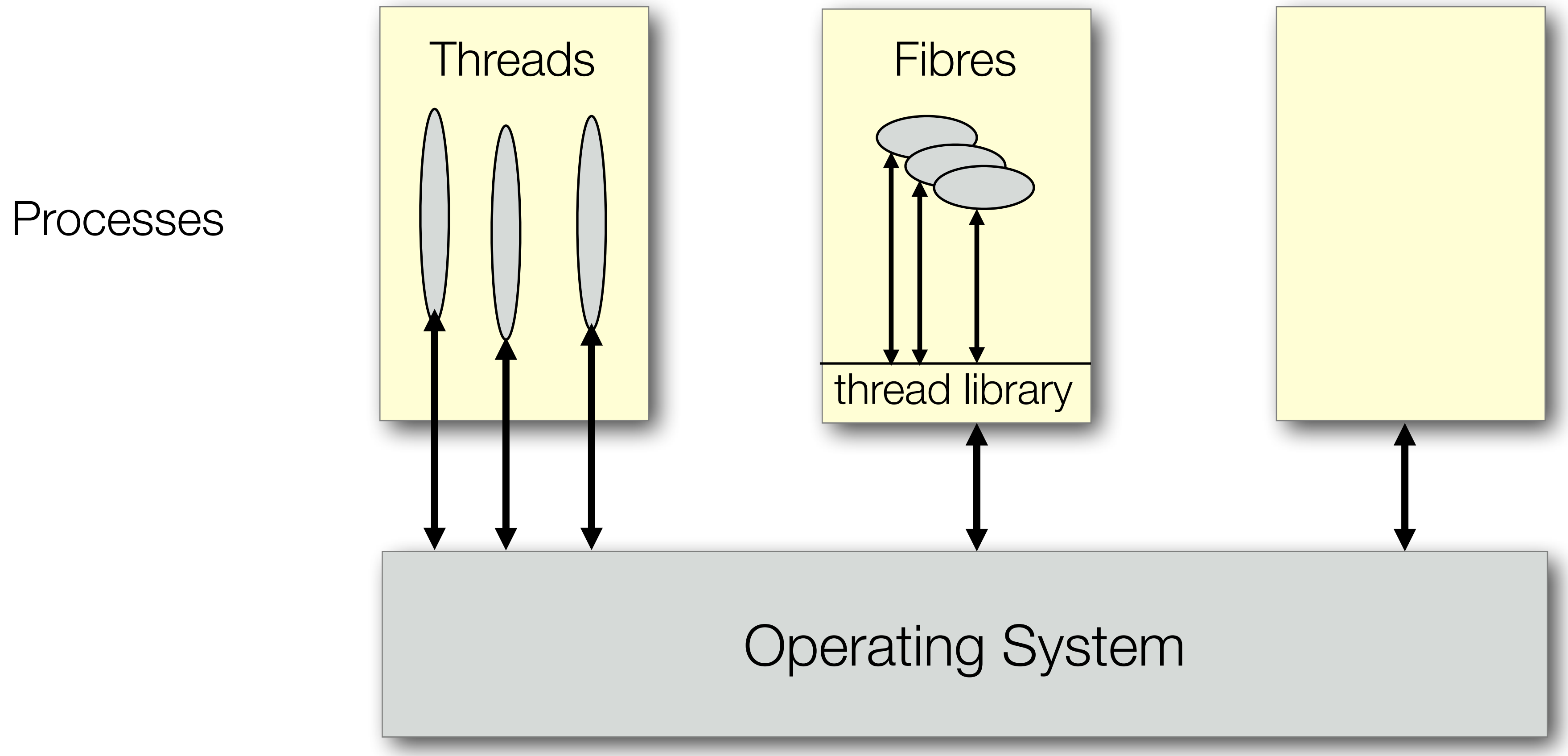
TECHNISCHE UNIVERSITÄT DARMSTADT

Concurrent Programming

(Overview of the Java concurrency model and its relationship to other models...)

(some slides are based on slides by Andy Wellings)

Processes vs. Threads (Concurrency Models)



"Threads" are sometimes called: lightweight processes; if so, "processes" are then called heavyweight processes.

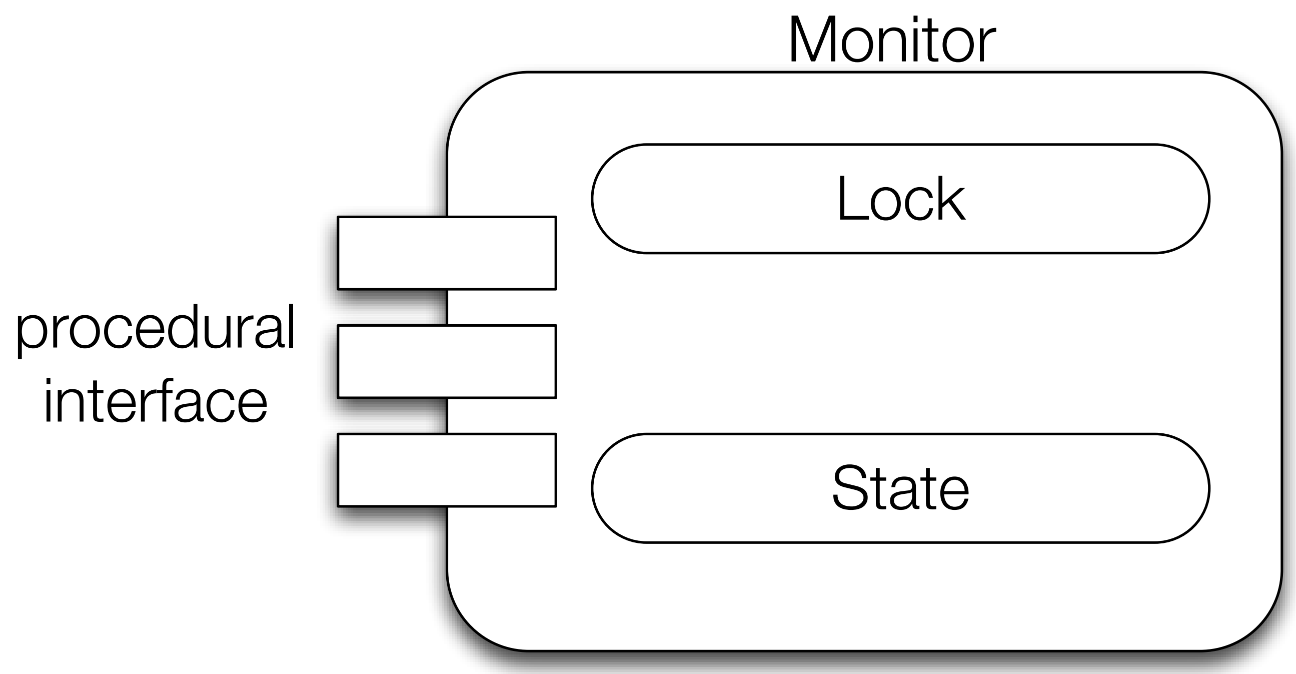
Fibres use cooperative multitasking

Java Supports Threads - Concurrency Models

- Threads execute within a single JVM.
- Types of Threads:
  - **Green threads** adopt the thread library approach
    (Green threads are invisible to the OS.)
  - **Native threads** map a single Java thread to an OS thread
    On a multiprocessor system, native threads are required to get true parallelism (but this is still implementation dependent).

Concurrency Models - Communication and Synchronization

- Communication by means of:
  - shared-variables (Java, C#, …)
  - message passing (Erlang, occam, process calculi,…)
- Many different models, a popular one is a **monitor**
  A monitor can be considered as an object where each of **its operation executes in mutual exclusion**.



Condition Synchronization

Concurrency Models

- ... expresses a constraint on the ordering of execution of operations,
- e.g., data cannot be removed from a buffer until data has been placed in the buffer.

Condition Synchronization

Concurrency Models

- *"Traditional Monitors"* provide multiple condition variables with two operations which can be called when the lock is held:
  - `wait`; an unconditional suspension of the calling thread (the thread is placed on a queue associated with the condition variable)
  - `notify`; one thread is taken from the queue associated with the respect condition variable and is re-scheduled for execution (it must reclaim the lock first)
  - `notifyAll`; all suspended threads are re-scheduled

(notify and notifyAll have no effect if no threads are suspended on the condition variable.)

Communication in Java Using Monitors

- Via *reading and writing to data encapsulated in shared objects* protected by (simple) monitors
- Every object is implicitly derived from the `Object` class which defines a mutual exclusion lock
- **Methods** in a class **can be labeled as synchronized**, this means that they can only be executed if the lock can be acquired (this happens automatically)
- The **lock can also be acquired via a synchronized statement** which names the object
- A thread can **wait** and **notify** on a single anonymous *condition variable*

Communication & Synchronization

Goals:

- To understand `synchronized` methods and statements and how they can be used with the `wait` and `notify` methods to implement simple monitors
- To show how to implement the bounded buffer communication paradigm

Synchronized Methods

- A mutual exclusion lock is (implicitly) associated with each object.
  The lock *cannot* be accessed directly by the application but is affected by:
  - the method modifier `synchronized`
  - block synchronization using the `synchronized` keyword
- When a method is labeled as **`synchronized`**, access to the method can only proceed once the system has obtained the lock
- Hence, **synchronized methods have mutually exclusive access to the data encapsulated by the object**, *if that data is only accessed by other synchronized methods*
- Non-synchronized methods do not require the lock and, therefore, can be called at any time

Double-Checked Locking Idiom

```
public class TACDemo {
    private static volatile TACDemo instance;
    static TACDemo getInstance() {
        TACDemo instance = TACDemo.instance;
        // thread-safe double checked locking
        if (instance == null) {
            synchronized (TACDemo.class) {
                instance = TACDemo.instance;
                if (instance == null) {
                    instance = new TACDemo();
                    TACDemo.instance = instance;
                }
            }
        }
        return instance;
    }
}
```

Synchronized Methods

Happens-before

- When a synchronized method exits, it establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object.
  - When the happens-before relation is established by a programmer, e.g., by means of synchronization, we have the guarantee that memory writes by statement A executed by Thread TA are visible to another specific statement B executed by Thread TB.

Example of Synchronized Methods

```
public class SharedInteger {

    private int theData;

    public SharedInteger(int initialValue) {
        theData = initialValue;
    }

    public synchronized int read() { return theData; }

    public synchronized void write(int newValue) { theData = newValue; }

    public synchronized void incrementBy(int by) {
        theData = theData + by;
    }
}

  ...
        SharedInteger myData = new SharedInteger(42);
```

... no synchronization of the constructor!
(not possible and not needed)

Synchronized Blocks

- A mechanism where a block can be labeled as **`synchronized`**
- The **`synchronized`** keyword takes as a parameter an object whose lock the system needs to obtain before it can continue
- Synchronized methods are effectively implementable as

```
public class SharedInteger {
   ...
   public int read() {
      synchronized (this) {
         return theData;
      }
   }
   ...
}
```

"this" is the Java mechanism for obtaining the current object.

Synchronized - Warning

- Used in its full generality, the synchronized block can undermine one of the advantages of monitor-like mechanisms, that of *encapsulating synchronization constraints associate with an object into a single place* in the program …
- … it is not possible to *understand the synchronization associated with a particular object by just looking at the object itself* when other objects can name that object in a synchronized statement
- However with careful use, this facility augments the basic model and allows more expressive synchronization constraints to be programmed

Accessing Synchronized Data

- Consider a simple class which implements a *two-dimensional coordinate* that is to be *shared between two or more threads*
- This class encapsulates two integers, whose values contain the **`x`** and the **`y`** coordinates
- Writing to a coordinate is simple, the write method can be labelled as `synchronized`
- Furthermore, the constructor method can be assumed not to have any synchronization constraint

Example of Synchronized Methods

```
public class SharedCoordinate {

    private int x, y;

    public SharedCoordinate(int initX, int initY) {
        x = initX;
        y = initY;
    }

    public synchronized void write(int newX, int newY) {
        x = newX;
        y = newY;
    }
    ...
}
```

… again, no synchronization of the constructor!

Accessing Synchronized Data

How to read the value of the coordinates?

- Functions in Java can only return a single value, and parameters to methods are passed by value
- Consequently, it is not possible to have a single `read` method which returns both the `x` and the `y` values
- If two synchronized functions are used, `readX` and `readY`, it is possible for the value of the coordinate to be written in between the calls to `readX` and `readY`
- The result will be an inconsistent value of the coordinate

Example of Synchronized Methods

Solution **Idea** 1

- Return a new coordinate object whose values of the x and y fields are identical to the shared coordinate
- This new object can then be accessed without fear of it being changed:

```java
public class SharedCoordinate {

   private int x, y;
  ...
   public synchronized SharedCoordinate read() {
      return new SharedCoordinate(x, y);
   }

   public int readX() { return x; }
   public int readY() { return y; }
}
```

Example of Synchronized Methods

Solution 1

Notes:

- The returned coordinate is only a **snapshot of the shared coordinate**, which might be changed by another thread immediately after the read method has returned
- The individual field values will be consistent
- Once the returned coordinate has been used, it can be discarded and made available for garbage collection
- If extreme efficiency is a concern, it is appropriate to try to avoid unnecessary object creation and garbage collection

Example of Synchronized Methods

Solution 2

- Assume the client thread will use synchronized blocks to obtain atomicity :

```
    ...
        SharedCoordinate point1 = new SharedCoordinate(0,0);
        synchronized (point1) {
            SharedCoordinate point2 = new SharedCoordinate(
                    point1.readX(),point1.readY());
        }
    ...
```

```
public class SharedCoordinate {
    private int x, y;
    ...
    public int readX() { return x; }
    public int readY() { return y; }
}
```

Static Data

- Static data is shared between all objects created from the class
- In Java, classes themselves are also objects and there is a **lock associated with the class**
- This lock may be accessed by either labeling a static method with the `synchronized` modifier or by identifying the class's object in a `synchronized` block statement
- The latter can be obtained from the `Object` class associate object
- Note that this class-wide lock is not obtained when synchronizing on the object

Using, e.g., “A.class”.

Static Data

```java
public class StaticSharedVariable {

    private static int globalCounter;

    public static int read() {
        synchronized (StaticSharedVariable.class) {
            return globalCounter;
        }
    }

    public synchronized static void write(int I) {
        globalCounter = I;
    }
}
```

Synchronization is required to make sure that you get the most recent version and not a cached version. (Without synchronization, the exact behavior is JVM dependent.)

Volatile

- Static and instances fields can be declared volatile; this ensures that all threads see consistent values (Java Memory Model)
- A write to a volatile field happens-before every subsequent read of that field.

Static Data

```
public class StaticSharedVariable {

    private static volatile int globalCounter;

    public static int getCounter() {
            return globalCounter;
    }

    public static void setCounter(int v) {
        globalCounter = v;
    }
}
```

Conditional Synchronization

Waiting and Notifying

- Conditional synchronization requires the methods provided in the predefined `Object` class:

```
...
public final void notify();

  public final void notifyAll();

  public final void wait() throws InterruptedException;

  public final void wait(long millis) throws InterruptedException;

  public final void wait(long millis, int nanos) throws InterruptedException;
  ...
```

Conditional Synchronization

Waiting and Notifying

- **These methods can be used only from within methods which hold the object lock**
- If called without the lock, the unchecked exception `IllegalMonitorStateException` is thrown
- The `wait` method always blocks the calling thread and releases the lock associated with the object

Conditional Synchronization

Waiting and Notifying

Notes:

- The `notify` method wakes up one waiting thread; the one woken is not defined by the Java language
- `notify` does not release the lock; hence the woken thread must wait until it can obtain the lock before proceeding
- To wake up all waiting threads requires use of the `notifyAll` method
- If no thread is waiting, then `notify` and `notifyAll` have no effect

Thread Interruption

- A waiting thread can also be awoken if it is interrupted by another thread
- In this case the `InterruptedException` is thrown (see later in the course)

Don’t do it!

Conditional Synchronization using Condition Variables

- There are no explicit condition variables in Java
- When a thread is awoken, it cannot assume that its condition is `true`, as all threads are *potentially awoken irrespective of what conditions they were waiting on*!
- For some algorithms this limitation is not a problem, as the conditions under which tasks are waiting are mutually exclusive
- ...

Conditional Synchronization using Condition Variables

- Example:
  - E.g., the bounded buffer traditionally has two condition variables: `BufferNotFull` and `BufferNotEmpty`
  - If a thread is waiting for one condition, no other thread can be waiting for the other condition
  - One would expect that the thread can assume that, when it awakes, the buffer is in the appropriate state
  - ...(to be continued)

Conditional Synchronization using Condition Variables

- Example:
  - ...(continued)
  - This is not always the case; Java makes no guarantee that a thread woken from a `wait` will gain immediate access to the lock
  - Another thread could call the put method, find that the buffer has space and insert data into the buffer
  - When the woken thread eventually gains access to the lock, the buffer will again be full
  - Hence, it is usually **essential for threads to re-evaluate their guards**

Bounded Buffer

```
public class BoundedBuffer {

    private final int buffer[];
    private int first;
    private int last;
    private int numberInBuffer = 0;
    private final int size;

  public BoundedBuffer(int length) {
        size = length;
        buffer = new int[size];
        last = 0;
        first = 0;
    };
    ...

}
```

Bounded Buffer

```
public class BoundedBuffer {
    ...
    public synchronized void put(int item) throws InterruptedException {
        while (numberInBuffer == size)
            wait();
        last = (last + 1) % size; // % is modulus
        numberInBuffer++;
        buffer[last] = item;
        notifyAll();
    };

    public synchronized int get() throws InterruptedException {
        while (numberInBuffer == 0)
            wait();
        first = (first + 1) % size; // % is modulus
        numberInBuffer--;
        notifyAll();
        return buffer[first];
    }
}
```

```
BoundedBuffer bb = new BoundedBuffer(1); bb.put(new Object()); // <= buffer is full!
…
Thread a = new Thread(new Runnable(){ public void run(){  … bb.put(new Object());… }}
Thread b = new Thread(new Runnable(){ public void run(){
                          …
                          bb.get();
                          …
                          bb.put(new Object()) ; }}
a.start(); b.start();
```

Executing Thread	method called	State of Thread "a"
a	bb.put(…)	buffer is full; a has to wait
b	bb.get()	bb's notifyAll() method is called; a is awoken
b	bb.put(…)	buffer is full; a is (still) ready
a	bb.put(…) is continued	buffer is full; a has to wait (again)

Concurrency - Why do we need to call “notifyAll” (and not notify)?

```
BoundedBuffer bb = new BoundedBuffer(1);
…
Thread g1,g2 = new Thread(){ public void run(){ bb.get(); } };
Thread p1,p2 = new Thread(){ public void run(){ bb.put(new Object()); } };
g1.start(); g2.start(); p1.start(); p2.start();
```

If `notify()` is used instead of `notifyAll()`.

	(concurrent) actions (“**bold**” = thread with the lock)	state of the buffer before and after the action	bb’s ready queue (Threads waiting for the lock.)	bb’s wait set (Sleeping Threads.)
1	**g1:bb.get()** g2:bb.get(), p1:bb.put(), p2:bb.put()	empty	{g2,p1,p2}	{g1}
2	**g2:bb.get()**	empty	{p1,p2}	scheduled {g1,g2}
3	**p1:bb.put()**	empty → not empty	{p2,g1}	{g2}
4	**p2:bb.put()**	not empty	{g1}	scheduled {g2,p2}
5	**g1:bb.get()**	not empty → empty	{g2}	{p2}
6	**g2:bb.get()**	empty	∅	{g2,~~p2~~}

Synchronization and Communication

Summary

- Errors in communication and synchronization cause working programs to suddenly suffer from ***deadlock*** or ***livelock***
- The Java model revolves around controlled access to shared data using a monitor-like facility
- The monitor is represented as an object with **`synchronized`** methods and statements providing mutual exclusion
- Condition synchronization is given by the **`wait`** and **`notify`** method
- True monitor condition variables are not directly supported by the language and have to be programmed explicitly

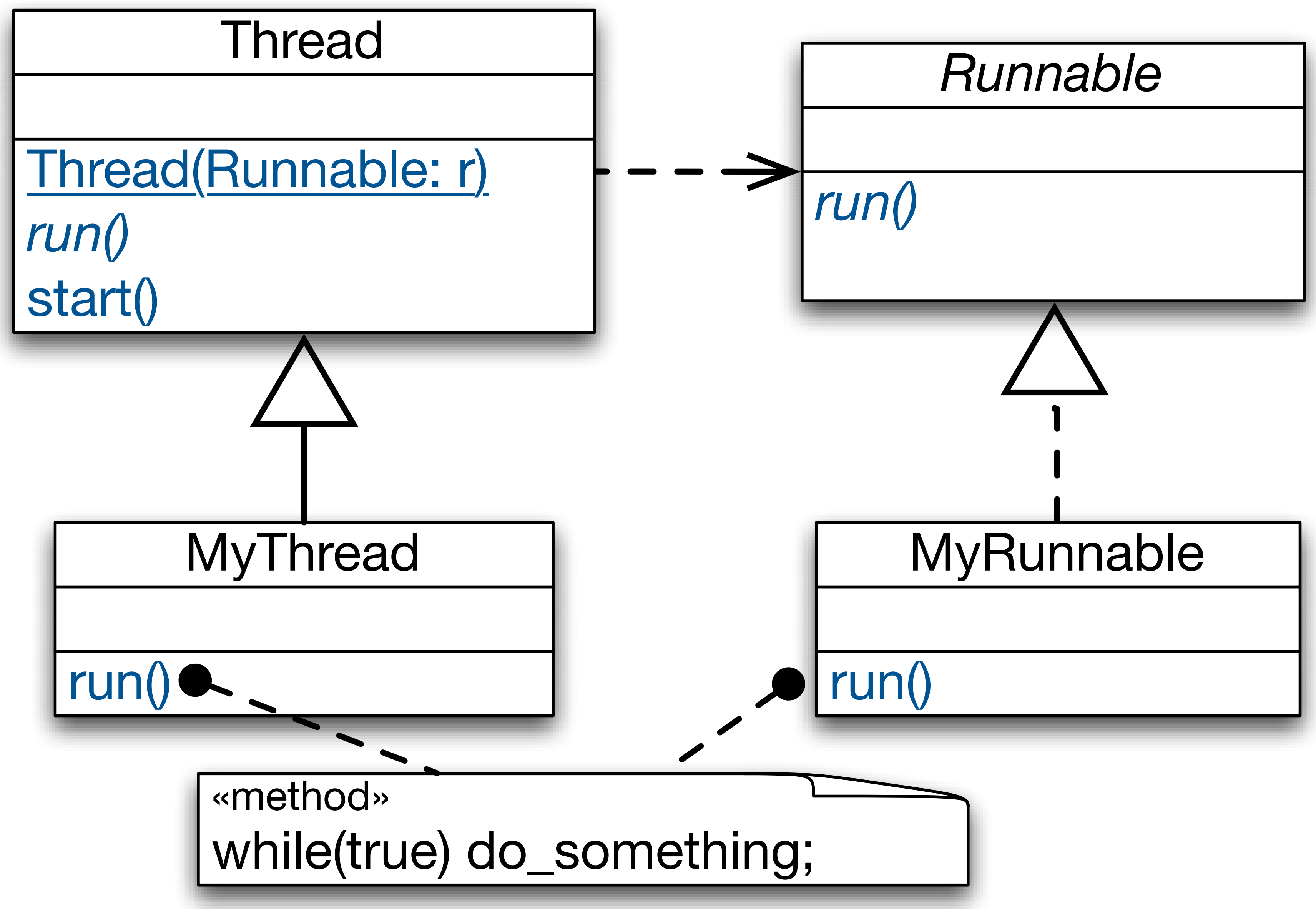
Concurrency in Java

- Java has a predefined class `java.lang.Thread` which provides the mechanism by which threads are created
- However to avoid all threads having to be child classes of `Thread`, it also uses a standard interface:

```
public interface Runnable {
  void run();
}
```

  - Hence, any class which wishes to express concurrent execution must implement this interface and provide the `run` method
  - Threads do not begin their execution until the `start` method in the `Thread` class is called

Threads in Java



java.lang.Thread

```
public class Thread implements Runnable {

    public Thread() {...}

    public Thread(Runnable target) {...}

    public Thread(ThreadGroup group, Runnable target) {...}

    public Thread(ThreadGroup group, Runnable target, String name,long stackSize) {...}

    ...

    public synchronized void start() {...}

    public void run() {...}

    ...
}
```

Thread Creation

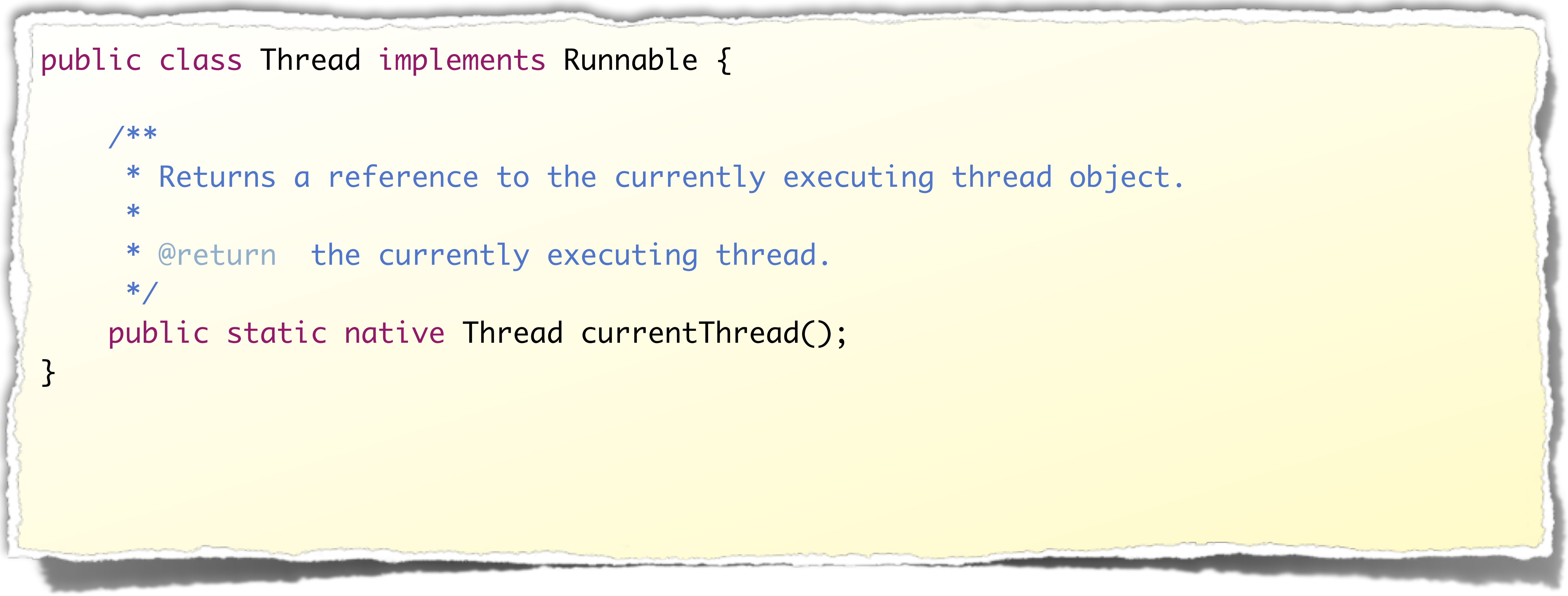
Two possibilities:

- Extend the `Thread` class and override the `run` method, or...
- Create an object which implements the `Runnable` interface and pass it to a `Thread` object via one of `Thread`’s constructors.

Thread Identification Using “currentThread()”

Identity of the currently running thread.

- `Thread.currentThread()` has a `static` modifier, which means that there is only one method for all instances of `Thread` objects
- The method can always be called using the `Thread` class



```
public class Thread implements Runnable {

    /**
     * Returns a reference to the currently executing thread object.
     *
     * @return  the currently executing thread.
     */
    public static native Thread currentThread();
}
```

Thread Termination

A Thread terminates...

- when it completes execution of its `run` method either normally or as the result of an unhandled exception
- via a call to its `stop` method — the `run` method is stopped and the thread class cleans up before terminating the thread (releases locks and executes any finally clauses)
  - The thread object is now eligible for garbage collection
  - Stop is inherently unsafe as it releases locks on objects and can leave those objects in inconsistent states; **the method is now deprecated and should not be used**
- by its `destroy` method being called — destroy terminates the thread without any cleanup (not provided by many JVMs, also deprecated)

Types of Threads

Java threads can be of two types: **user threads** or **daemon threads**

- Daemon threads are those threads which provide general services and typically never terminate
- When all user threads have terminated, daemon threads can also be terminated and the main program terminates
- The `setDaemon` method must be called before the thread is started

Joining Threads

Inter-thread Communication

- One thread can wait (with or without a timeout) for another thread (the target) to terminate by issuing the `join` method call on the target's thread object
- The `isAlive` method allows a thread to determine if the target thread has terminated

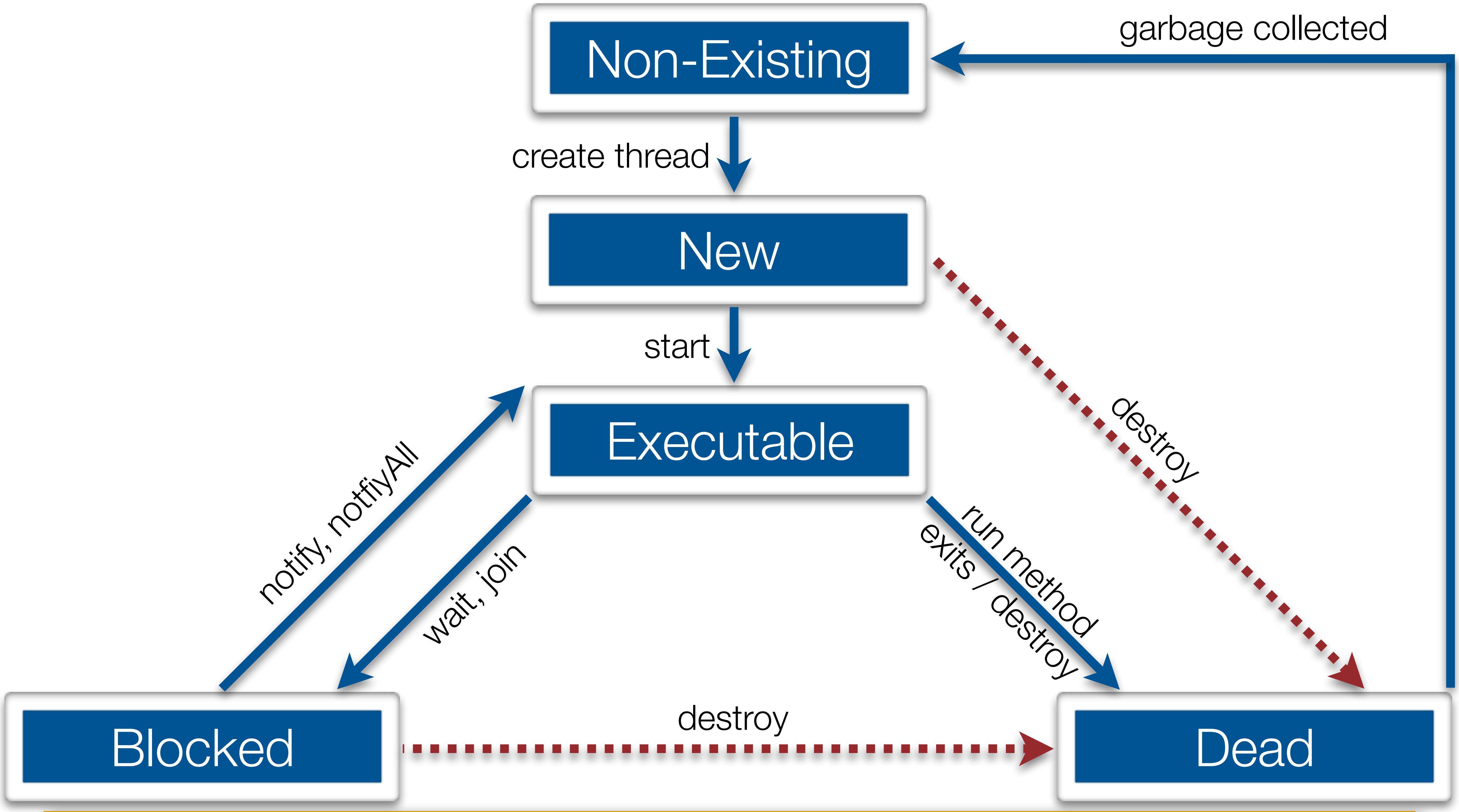
java.lang.Thread

Inter-thread Communication

```
public class Thread implements Runnable {
    ...

    public final native boolean isAlive() {...}

    public final native void join() throws InterruptedException {...}

    public final native void join(long millis) throws InterruptedException {...}

    public final native void join(long millis, int nanos)
            throws InterruptedException {...}
    ...
}
```

Note, there is no guarantee, that your thread will be waked up, after the specified time! It is only guaranteed, that the thread - at least - sleeps for the specified time (unless it is destroyed or otherwise interrupted.)

Java Thread States



http://download.oracle.com/javase/6/docs/api/java/lang/Thread.State.html

Java Thread States

Summary

- The thread is created when an object derived from the `Thread` class is created
- At this point, the thread is not executable — Java calls this the **new** state
- Once the `start` method has been called, the thread becomes **eligible for execution** by the scheduler
- If the thread calls the `wait` method in an object, or calls the `join` method on another thread object, the thread becomes **blocked** and is no longer eligible for execution
- The thread becomes executable as a result of an associated `notify` method being called by another thread, or if the thread with which it has requested a join, becomes **dead**

Java Thread States

Summary

- A thread enters the **dead** state, either as a result of the `run` method exiting (normally or as a result of an unhandled exception) or because its `destroy` method has been called
- In the latter case, the thread is abruptly moved to the **dead** state and does not have the opportunity to execute any finally clauses associated with its execution; it may leave other objects locked

Synchronization and Communication

Java 1.5 Concurrency API

Java 1.5 Concurrency Utilities

Support for general-purpose concurrent programming.

- `java.util.concurrent`
  Provides various classes to support common concurrent programming paradigms, e.g., support for various queuing policies such as bounded buffers, sets and maps, thread pools etc.
- `java.util.concurrent.atomic`
  Provides support for *lock-free thread-safe programming on simple variables* such as atomic integers, atomic booleans, etc.
- `java.util.concurrent.locks`
  Provides a framework for various *locking algorithms that augment the Java* language mechanisms, e.g., read -write locks and condition variables.

Java 1.5 Locks

Support for general-purpose concurrent programming.

Lock implementations provide more extensive and more sophisticated locking operations than can be obtained using synchronized methods and statements.

- For example, 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

Java 1.5 Locks

Support for general-purpose concurrent programming.

- “hand-over-hand” or “chain locking” require more flexible locks
  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.
- 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.

Java 1.5 Locks

Support for general-purpose concurrent programming.

- Additional functionality over the use of synchronized methods and statements: non-blocking attempt to acquire a lock (`tryLock()`), an attempt to acquire the lock that can be interrupted (`lockInterruptibly()`), and an attempt to acquire the lock that can timeout (`tryLock(long, TimeUnit)`)
- A `Lock` class can also provide behavior and semantics that is quite different from that of the implicit monitor lock, such as guaranteed ordering, non-reentrant usage, or deadlock detection

Java 1.5 Locks

java.util.concurrent.locks

```java
public interface Lock {

    /** Wait for the lock to be acquired. */
    public void lock();

    /** Create a new condition variable for use with the Lock. */
    public Condition newCondition();

    public void unlock();
}
```

Java 1.5 Conditions (w.r.t. Locks)

Support for general-purpose concurrent programming.

- A condition factors out the `Object` monitor methods (`wait`, `notify` and `notifyAll`) into distinct objects to give the effect of having multiple wait-sets per object, by combining them with the use of arbitrary `Lock` implementations
- Where a `Lock` replaces the use of synchronized methods and statements, a `Condition` replaces the use of the `Object` monitor methods
- A `Condition` instance is *intrinsically bound to a lock*
  To obtain a `Condition` instance for a particular Lock instance use its `newCondition()` method.

Java 1.5 Locks

java.util.concurrent.locks

```
public interface Condition {

   /**
    * Atomically releases the associated lock and causes the current thread to
    * wait.
    */
   public void await() throws InterruptedException;


   /** Wake up one waiting thread. */
   public void signal();


   /** Wake up all waiting threads. */
   public void signalAll();

}
```

Java 1.5 Locks

java.util.concurrent.locks

```
public class ReentrantLock implements Lock {

    public ReentrantLock() {...}

    public void lock() {...}

    public void unlock() {...}

    /**
     * Create a new condition variable and associated it with this lock object.
     */
    public Condition newCondition() {...}

}
```

Generic Bounded Buffer - State

```
public class BoundedBuffer<T> {

   private final T buffer[];

   private int first;

   private int last;

   private int numberInBuffer;

   private final int size;

   private final Lock lock = new ReentrantLock();

   private final Condition notFull = lock.newCondition();

   private final Condition notEmpty = lock.newCondition();

  ...
}
```

The “usual” variables required to implement a bounded buffer.

Two Condition variables: one for the case that no data is stored in the buffer and one for the case that the buffer is full.

Generic Bounded Buffer - Constructor

```
public class BoundedBuffer<T> {

  ...

    public BoundedBuffer(int length) {

        size = length;
        buffer = (T[]) new Object[size];
        last = 0;
        first = 0;
        numberInBuffer = 0;
    }

    ...
}
```

Nothing special w.r.t. handling concurrency.

Generic Bounded Buffer - Putting Data in the Buffer

```
public class BoundedBuffer<T> {

  ...
    public void put(T item) throws InterruptedException {
        lock.lock();
        try {

            while (numberInBuffer == size) { notFull.await(); }
            last = (last + 1) % size;
            numberInBuffer++;
            buffer[last] = item;
            notEmpty.signal();

        } finally {
            lock.unlock();
        }
    }
    ...
}
```

At least one entry is now stored in the buffer, so we

Important idiom: always release the

Comparison of Both Bounded Buffer Implementations

Getting Data

```
public class BoundedBuffer<T> {

  ...
    public T get() ... {
        lock.lock();
        try {

            while (numberInBuffer == 0)
                { notEmpty.await(); }
            first = (first + 1) % size;
            numberInBuffer--;
            notFull.signal();
            return buffer[first];

        } finally {
            lock.unlock();
        }
    }
}
```

```
public class BoundedBuffer<T> {
    ...

    public synchronized T get() ... {



        while (numberInBuffer == 0)
            wait();
        first = (first + 1) % size;
numberInBuffer--;
        notifyAll();
        return buffer[first];





    }
}
```

Best Practices

- Synchronized code should be kept as short as possible
- Nested monitor calls:
  - ... should be avoided because **the outer lock is not released when the inner monitor waits** (to release the lock causes other problems).
  - … can easily lead to deadlock occurring
  - … (continued on the next slide.)

Best Practices

- Nested monitor calls:
  - … (continues the previous slide.)
  - It is not always obvious when a nested monitor call is being made:
    - ... methods not labelled as `synchronized` can still contain a `synchronized` statement
    - ... methods in a class not labelled as `synchronized` can be overridden with a `synchronized` method; method calls which start off as being un-`synchronized` may be used with a `synchronized` subclass
    - ... methods called via interfaces cannot be labelled as `synchronized`

Thread Safety

What does thread safety mean?

Prerequisites:

- For a class to be thread-safe, it must behave correctly in a single-threaded environment
- If a class is correctly implemented, no sequence of operations (reads or writes of non-private fields and calls to non-private methods) on objects of that class should be able to:
  - put the object into an invalid state,
  - observe the object to be in an invalid state, or
  - violate any of the class's invariants, preconditions, or postconditions.

Thread Safety

What does thread safety mean?

- For a class to be thread-safe, it must continue to behave correctly, in the sense described on the previous slide,...
  - when accessed from multiple threads regardless of the scheduling or interleaving of the execution of those threads by the runtime environment,
  - without any additional synchronization on the part of the calling code

The effect is that **operations on a thread-safe object will appear to all threads to occur in a fixed, globally consistent order**.

Bloch’s Thread Safety Levels

- **Immutable**
  Objects are constant and cannot be changed
- **Thread-safe**
  Objects are mutable, but they can be used safely in a concurrent environment as the methods are appropriately synchronized
- **Conditionally thread-safe**
  Conditionally thread-safe classes are those for which each individual operation may be thread-safe, but certain sequences of operations may require external synchronization
  E.g.: traversing an `Iterator` returned from `Hashtable` or `Vector`. The fail-fast iterators returned by these classes assume that the underlying collection will not be mutated while the iterator traversal is in progress.
- …

Bloch’s Thread Safety Levels

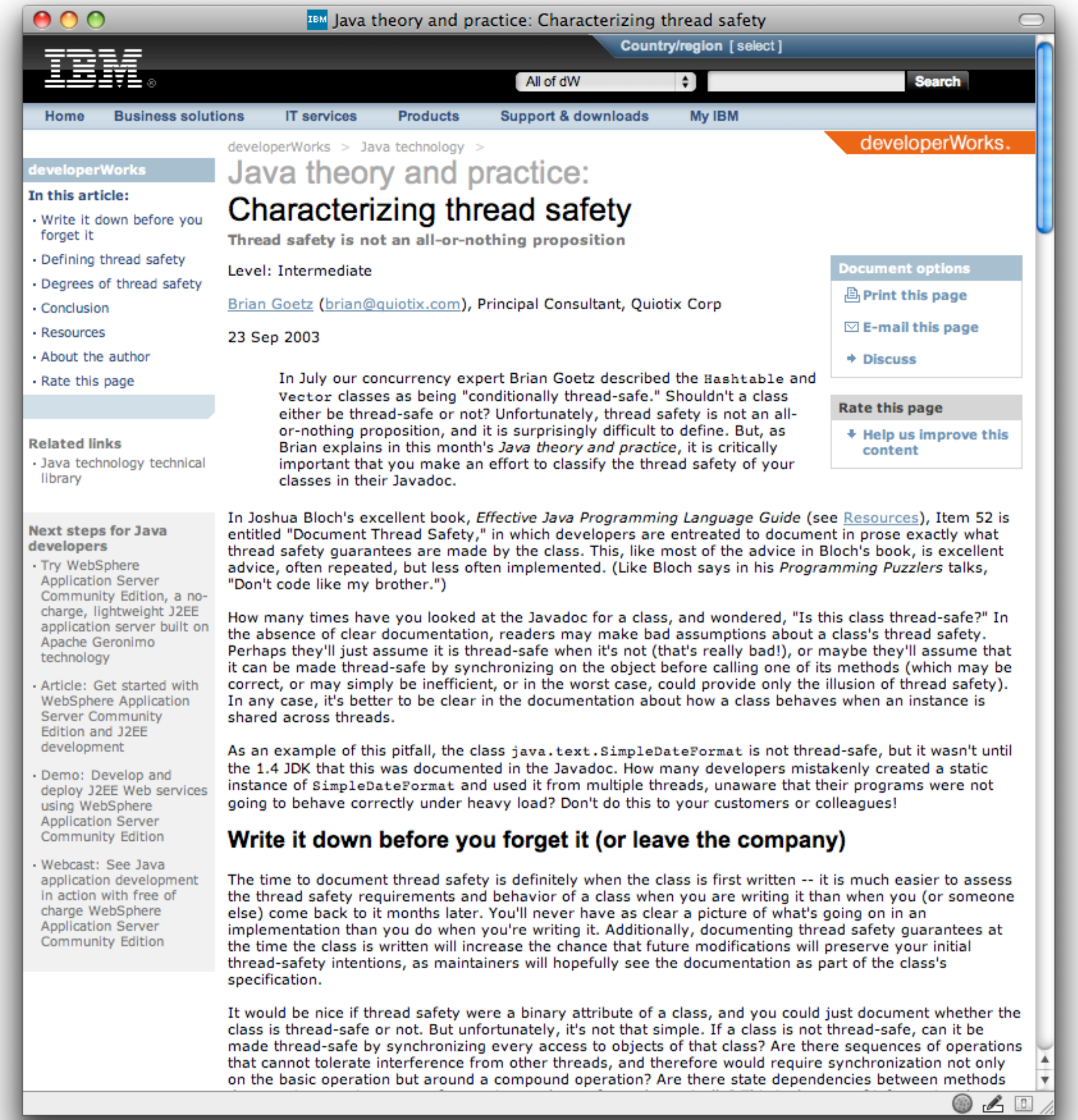
- …
- **Thread compatible**
  - Instances of the class provide no synchronization
  - However, instances of the class can be safely used in a concurrent environment, if the caller provides the synchronization by surrounding each method (or sequence of method calls) with the appropriate lock
- …

Bloch’s Thread Safety Levels

- …
- **Thread-hostile**
  - Instances of the class should not be used in a concurrent environment even if the caller provides external synchronization
  - Typically a thread hostile class is accessing static data or the external environment
  - An example of a thread-hostile class would be one that calls `System.setOut()`

Bloch’s Thread Safety Levels

recommended reading (a very concise summary)



developerWorks > Java technology >

Java theory and practice:

Characterizing thread safety

Thread safety is not an all-or-nothing proposition

Level: Intermediate

Brian Goetz (brian@quiotix.com), Principal Consultant, Quiotix Corp

23 Sep 2003

In July our concurrency expert Brian Goetz described the `Hashtable` and `Vector` classes as being "conditionally thread-safe." Shouldn't a class either be thread-safe or not? Unfortunately, thread safety is not an all-or-nothing proposition, and it is surprisingly difficult to define. But, as Brian explains in this month's *Java theory and practice*, it is critically important that you make an effort to classify the thread safety of your classes in their Javadoc.

developerWorks

In this article:
- Write it down before you forget it
- Defining thread safety
- Degrees of thread safety
- Conclusion
- Resources
- About the author
- Rate this page

Related links
- Java technology technical library

Next steps for Java developers
- Try WebSphere Application Server Community Edition, a no-charge, lightweight J2EE application server built on Apache Geronimo technology
- Article: Get started with WebSphere Application Server Community Edition and J2EE development
- Demo: Develop and deploy J2EE Web services using WebSphere Application Server Community Edition
- Webcast: See Java application development in action with free of charge WebSphere Application Server Community Edition

Document options
- Print this page
- E-mail this page
- Discuss

Rate this page
- Help us improve this content

In Joshua Bloch's excellent book, *Effective Java Programming Language Guide* (see Resources), Item 52 is entitled "Document Thread Safety," in which developers are entreated to document in prose exactly what thread safety guarantees are made by the class. This, like most of the advice in Bloch's book, is excellent advice, often repeated, but less often implemented. (Like Bloch says in his *Programming Puzzlers* talks, "Don't code like my brother.")

How many times have you looked at the Javadoc for a class, and wondered, "Is this class thread-safe?" In the absence of clear documentation, readers may make bad assumptions about a class's thread safety. Perhaps they'll just assume it is thread-safe when it's not (that's really bad!), or maybe they'll assume that it can be made thread-safe by synchronizing on the object before calling one of its methods (which may be correct, or may simply be inefficient, or in the worst case, could provide only the illusion of thread safety). In any case, it's better to be clear in the documentation about how a class behaves when an instance is shared across threads.

As an example of this pitfall, the class `java.text.SimpleDateFormat` is not thread-safe, but it wasn't until the 1.4 JDK that this was documented in the Javadoc. How many developers mistakenly created a static instance of `SimpleDateFormat` and used it from multiple threads, unaware that their programs were not going to behave correctly under heavy load? Don't do this to your customers or colleagues!

Write it down before you forget it (or leave the company)

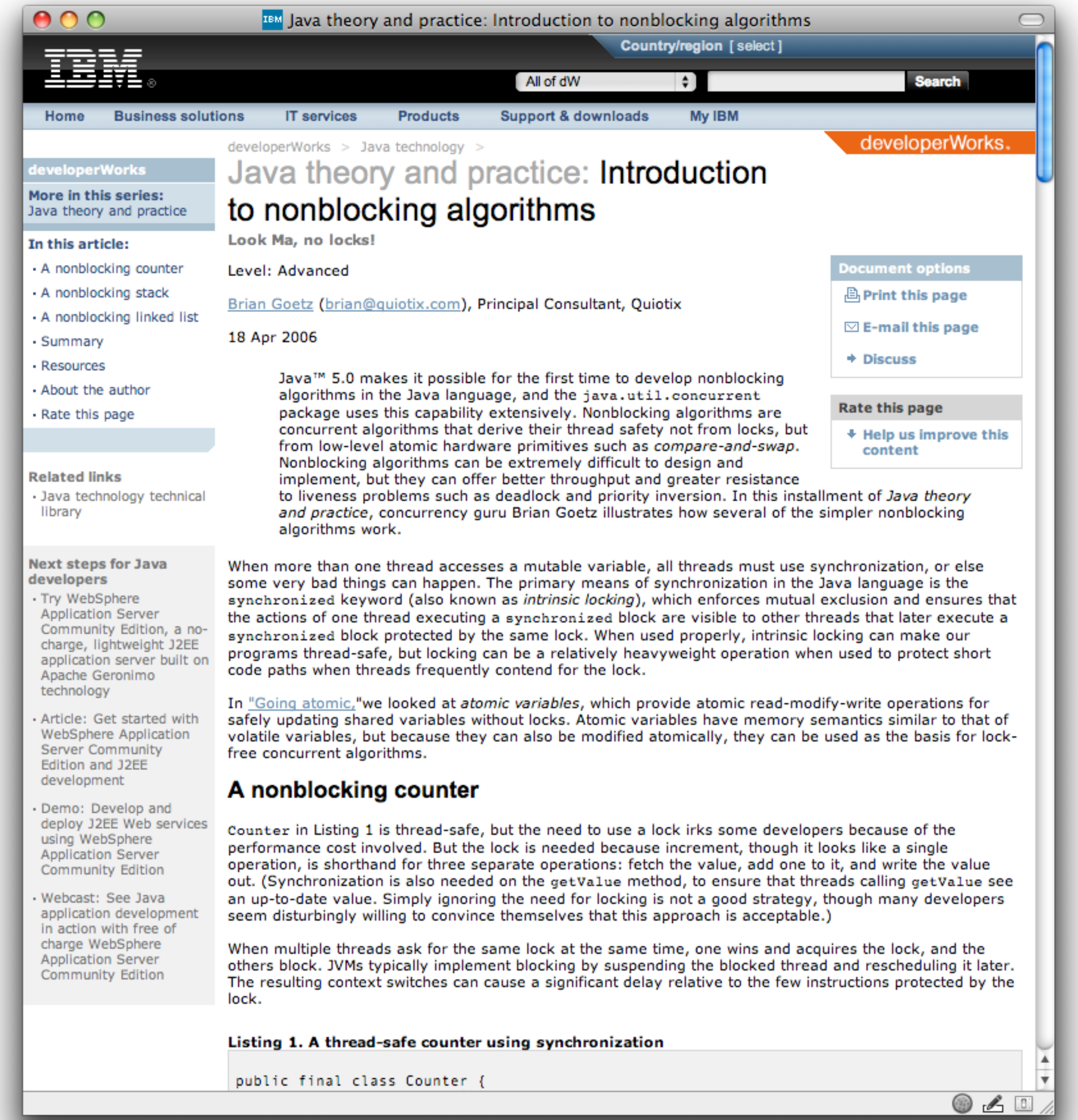
The time to document thread safety is definitely when the class is first written -- it is much easier to assess the thread safety requirements and behavior of a class when you are writing it than when you (or someone else) come back to it months later. You'll never have as clear a picture of what's going on in an implementation than you do when you're writing it. Additionally, documenting thread safety guarantees at the time the class is written will increase the chance that future modifications will preserve your initial thread-safety intentions, as maintainers will hopefully see the documentation as part of the class's specification.

It would be nice if thread safety were a binary attribute of a class, and you could just document whether the class is thread-safe or not. But unfortunately, it's not that simple. If a class is not thread-safe, can it be made thread-safe by synchronizing every access to objects of that class? Are there sequences of operations that cannot tolerate interference from other threads, and therefore would require synchronization not only on the basic operation but around a compound operation? Are there state dependencies between methods

http://www.50001.com/language/javaside/lec/java_ibm/%BE%B2%B7%B9%B5%E5%20%BA%B8%BE%C8%20(%BF%B5%B9%AE).htm

Nonblocking Algorithms

Further Information



developerWorks > Java technology >

Java theory and practice: Introduction to nonblocking algorithms

Look Ma, no locks!

Level: Advanced

Brian Goetz (brian@quiotix.com), Principal Consultant, Quiotix

18 Apr 2006

More in this series:
Java theory and practice

In this article:
- A nonblocking counter
- A nonblocking stack
- A nonblocking linked list
- Summary
- Resources
- About the author
- Rate this page

Related links
- Java technology technical library

Next steps for Java developers
- Try WebSphere Application Server Community Edition, a no-charge, lightweight J2EE application server built on Apache Geronimo technology
- Article: Get started with WebSphere Application Server Community Edition and J2EE development
- Demo: Develop and deploy J2EE Web services using WebSphere Application Server Community Edition
- Webcast: See Java application development in action with free of charge WebSphere Application Server Community Edition

Java™ 5.0 makes it possible for the first time to develop nonblocking algorithms in the Java language, and the `java.util.concurrent` package uses this capability extensively. Nonblocking algorithms are concurrent algorithms that derive their thread safety not from locks, but from low-level atomic hardware primitives such as *compare-and-swap*. Nonblocking algorithms can be extremely difficult to design and implement, but they can offer better throughput and greater resistance to liveness problems such as deadlock and priority inversion. In this installment of *Java theory and practice*, concurrency guru Brian Goetz illustrates how several of the simpler nonblocking algorithms work.

When more than one thread accesses a mutable variable, all threads must use synchronization, or else some very bad things can happen. The primary means of synchronization in the Java language is the `synchronized` keyword (also known as *intrinsic locking*), which enforces mutual exclusion and ensures that the actions of one thread executing a `synchronized` block are visible to other threads that later execute a `synchronized` block protected by the same lock. When used properly, intrinsic locking can make our programs thread-safe, but locking can be a relatively heavyweight operation when used to protect short code paths when threads frequently contend for the lock.

In "Going atomic,"we looked at *atomic variables*, which provide atomic read-modify-write operations for safely updating shared variables without locks. Atomic variables have memory semantics similar to that of volatile variables, but because they can also be modified atomically, they can be used as the basis for lock-free concurrent algorithms.

A nonblocking counter

`Counter` in Listing 1 is thread-safe, but the need to use a lock irks some developers because of the performance cost involved. But the lock is needed because increment, though it looks like a single operation, is shorthand for three separate operations: fetch the value, add one to it, and write the value out. (Synchronization is also needed on the `getValue` method, to ensure that threads calling `getValue` see an up-to-date value. Simply ignoring the need for locking is not a good strategy, though many developers seem disturbingly willing to convince themselves that this approach is acceptable.)

When multiple threads ask for the same lock at the same time, one wins and acquires the lock, and the others block. JVMs typically implement blocking by suspending the blocked thread and rescheduling it later. The resulting context switches can cause a significant delay relative to the few instructions protected by the lock.

Listing 1. A thread-safe counter using synchronization

```
public final class Counter {
```

to get started: http://www.ibm.com/developerworks/library/j-jtp04186/