Introduction to Concurrent Programming in Java™
(lots of slides cut for 433)

David Holmes
DLTeCH Pty Ltd
Brisbane, Australia
dholmes@dlttech.com.au

Doug Lea
State University of New York
Oswego, NY
dl@cs.oswego.edu
http://gee.cs.oswego.edu/~dl/

Designing Objects for Concurrency

- Isolation
  - Avoiding interference by not sharing
- Immutability
  - Avoiding interference by avoiding change
- Locking
  - Dynamically guaranteeing exclusive access
- Splitting objects
  - Changing representation to facilitate concurrency control
- Containment
  - Guaranteeing exclusive control of internal components
  - Managing ownership
  - Protecting unhidden components
- Alternatives to synchronization
  - volatiles and the Java Memory Model
Isolation

Objects that are not shared can not suffer interference

- Heap objects accessible only from current thread
- Parameters and local variables
  - Applies to references not the objects referred to
- `java.lang.ThreadLocal`
  - Simplifies access from other objects running in same thread
- No need for any synchronization

Objects can be shared across threads provided they are isolated to one thread at a time

- Transfer of ownership protocols
  - T1 uses O1, hands off to T2 and then forgets about O1
  - Transfer requires synchronization—subsequent use of object does not

Thread Locals

- Suppose you want multiple web servers, each running in a different thread, and each using a different document directory
  - Could define a `documentRoot` field in `WebServer` class
- Or, define the document root as a variable tied to the `Thread`
  - Easiest way to do this is to use `java.lang.ThreadLocal`
  - Equivalent to adding instance variables to all `Thread` objects

  - No need to define subclasses or control thread creation
  - All methods running in the thread can access when needed
    - `ThreadLocals` are often package accessible statistics
  - No interference when ALL access is within same thread

    ```java
    public class WebServer {
        static final ThreadLocal documentRoot = new ThreadLocal();
        // ...
        public WebServer(int port, File root) throws IOException {
            // ...
            documentRoot.set(root);
        }
        private void processRequest(Socket sock) throws IOException {
            File root = (File) documentRoot.get();
        }
    }
    ```
**When to Use Thread Locals**

- Variables that apply per-activity, not per-object
  - Timeout values, transaction IDs, Principals, current directories, default parameters
- Replacements for static variables
  - When different threads should use different values
- Tools to eliminate need for locking
  - Used internally in JVMs to optimize memory allocation, locks, etc via per-thread caches

**Stateless Objects**

```java
class StatelessAdder {
    int addOne( int i) { return i + 1; }
    int addTwo( int i) { return i + 2; }
}
```

- There are no special concurrency concerns
  - No storage conflicts as no per-instance state
  - No representation invariants as no representation
  - Multiple concurrent executions—so no liveness problems
  - No need to create threads to make this call
  - No interaction with other objects—so no concurrent protocol design issues
- Example: `java.lang.Math`
Immutable Objects

class ImmutableAdder {
    private final int offset;

    ImmutableAdder(int offset) { this.offset = offset; }

    int add(int i) { return i + offset; }
}

- Object state frozen upon initialisation
  - Still no safety or liveness concerns
  - No interference as per-instance state never changes
  - Java finals enforce most senses of immutability

- Immutability is often suitable for closed Abstract Data Types eg.
  - java.lang.String, java.lang.Integer

Containment of Unsafe Objects

- Suppose Statistics class was written as follows:

  ```java
  public static class Statistics {
      // Mutable!
      public long requests;
      public double avgTime;
      public Statistics(long requests, double avgTime) {
          this.requests = requests; this.avgTime = avgTime;
      }
  }
  ```
  - Fields are public and mutable!
    - Therefore instances can not be shared

- Can be safely contained within a WebServer instance

  ```java
  class WebServer {
      private final Statistics stats = new Statistics(0,0.0);
      public synchronized Statistics getStatistics() {
          return new Statistics(stats.requests, stats.avgTime);
      }

      private void processRequest(Socket sock) throws IOException {
          synchronized(this) {
              double total = stats.avgTime*stats.requests + elapsed;
              stats.avgTime = total / (++stats.requests);
          }
      }
  }
  ```
  - Can’t expose mutable state so we make copies of it
Containment

- Strict containment creates islands of objects
  - Applies recursively
- Allows inner code to run faster
  - Can be used with legacy sequential code
- Requires inner code to be communication closed
  - No unprotected calls into or out of island
- Requires outer objects to never leak inner references
  - Or uses ownership transfer protocol
- Can be difficult to enforce and check

Hierarchical Containment Locking

- Applies when logically contained parts are not hidden from clients
- Avoids deadlocks that could occur if parts fully synchronised
- All parts use lock provided by the common owner
  - Can use either internal or external conventions
Internal Containment Locking

- Visible components protect themselves using their owners' locks
  ```java
  class Part {
      protected Container owner_; // Never null
      public Container owner() { return owner_; }
      private void bareAction() { /* ... unsafe ... */ }
      public void m() {
          synchronized (owner()) { bareAction(); }
      }
  }
  ``
  - Parts don't deadlock when invoking each other's methods
  - Parts must be aware that they are contained
  - Or implement using inner classes—Owner is outer class:
    ```java
    class Container {
        class Part {
            // ...
            public void m() {
                synchronized (Container.this) { bareAction(); }
            }
        }
    }
    ``
  - Can extend to frameworks based on shared Lock objects, transaction locks, etc rather than `synchronized` blocks

External Containment Locking

- Rely on callers to provide the locking
  - Client-side synchronization
    ```java
    class Client {
        void f(Part p) {
            synchronized (p.owner()) { p.bareAction(); }
        }
    }
    ``
  - Used in AWT
    ```java
    java.awt.Component.getTreeLock()
    ``
  - Can sometimes avoid more locking overhead, at price of fragility
    - Can manually minimize use of `synchronized`
    - Requires that all callers obey conventions
    - Effectiveness is context dependent
      - Breaks encapsulation
      - Doesn't work with fancier schemes that do not directly rely on `synchronized` blocks or methods for locking
Subclassing Unsafe Code

Suppose `processRequest` invokes

```java
handlerHelper.mountFileSystem();
```

where:

```java
class HandlerHelper{
    native void mountFileSystem();
}
```

If we don’t trust this class to be thread-safe, we could

- Wrap calls in synch blocks (i.e., containment), or
- Create a simple subclass that adds synch...

```java
class SafeHandlerHelper extends HandlerHelper{
    synchronized void mountFileSystem() {
        super.mountFileSystem();
    }
}
```

... and instantiate it instead

- This localizes synch control in the place it is needed

Subclassing is usually the most convenient way to do this

- Can also use unrelated wrapper classes and delegate
- Can generalize to “template method” schemes (discussed later)

State Dependent Actions

- State Dependence
- Balking
- Guarded Suspension
- Optimistic Retries
- Specifying Policies
Examples of State Dependent Actions

- Operations on collections, streams, databases
  - Remove an element from an empty queue
  - Add an element to a full buffer
- Operations on objects maintaining constrained values
  - Withdraw money from an empty bank account
- Operations requiring resources
  - Print a file
- Operations requiring particular message orderings
  - Read an unopened file
- Operations on external controllers
  - Shift to reverse gear in a moving car

Policies for State Dependent Actions

- Some Policy choices for dealing with pre-and post-conditions

<table>
<thead>
<tr>
<th>Policy</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blind action</td>
<td>Proceed anyway; no guarantee of outcome</td>
</tr>
<tr>
<td>Inaction</td>
<td>Ignore request if not in right state</td>
</tr>
<tr>
<td>Balking</td>
<td>Fail (throw exception) if not in right state</td>
</tr>
<tr>
<td>Guarding</td>
<td>Suspend until in right state</td>
</tr>
<tr>
<td>Trying</td>
<td>Proceed, check if succeeded; if not, roll back</td>
</tr>
<tr>
<td>Retrying</td>
<td>Keep trying until success</td>
</tr>
<tr>
<td>Timing out</td>
<td>Wait or retry for a while; then fail</td>
</tr>
<tr>
<td>Planning</td>
<td>First initiate activity that will achieve right state</td>
</tr>
</tbody>
</table>
Interfaces and Policies

```java
public interface Buffer {
    int capacity();  // Inv: capacity() > 0
    int size();      // Inv: 0 ≤ size() ≤ capacity()
                     // Init: size() == 0
    void put(Object x); // Pre: size() < capacity()
    Object take();    // Pre: size() > 0
}
```

- Interfaces alone cannot convey policy
- But can suggest policy
  - For example, should `take()` throw exception? What kind?
  - Different methods can support different policies for same base actions
- But can use manual annotations
  - Declarative constraints form basis for implementation
- For examples we throw `Failure`:
  ```java
  class Failure extends Exception {...}
  ```

Balking

- Check state upon method entry
  - Must not change state in course of checking it
  - Relevant state must be explicitly represented, so can be checked upon entry
- Exit immediately if not in right state
  - Throw exception or return special error value
  - Client is responsible for handling failure
- The simplest policy for fully synchronized objects
  - Useable in both sequential and concurrent contexts
    - Often used in Collection classes (`Vector`, etc)
  - In concurrent contexts, the host must always take responsibility for entire check-act/check-fail sequence
    - Clients cannot preclude state changes between check and act, so host must control
Example: Balking Bounded Buffer

```java
public class BalkingBoundedBuffer implements Buffer {
    private List data;
    private final int capacity;
    public BalkingBoundedBuffer(int capacity) {
        data = new ArrayList(capacity);
        this.capacity = capacity;
    }
    public synchronized Object take() throws Failure {
        if (data.size() == 0)
            throw new Failure("Buffer empty");
        Object temp = data.get(0);
        data.remove(0);
        return temp;
    }
    public synchronized void put(Object obj) throws Failure {
        if (data.size() == capacity)
            throw new Failure("Buffer full");
        data.add(obj);
    }
    public synchronized int size() { return data.size();}
    public int capacity() {return capacity;}
}
```

Guarding

- Generalisation of locking for state dependent actions
  - **Locked**: Wait until ready (not engaged in other methods)
  - **Guarded**: Wait until an arbitrary state predicate holds

- Check state upon entry
  - If not in right state, wait
  - Some other action in some other thread may eventually cause a state change that enables resumption

- Introduces liveness concerns
  - Relies on actions of other threads to make progress

- Useless in sequential programs
  - Client must ensure correct state before calling
Guarding Mechanisms

- **Busy-waits**
  - Thread continually spins until a condition holds
    ```java
    while (!condition); // spin
    // use condition
    ```
  - Requires multiple CPU's or timeslicing
    - No way to determine this until JDK 1.4
      ```java
      int nCPUs = Runtime.availableProcessors();
      ```
  - But busy waiting can sometimes be useful; generally when
    - The conditions **latch**—once set true, they never become false

- **Suspension**
  - Thread **stops execution** until notified that the condition **may** be true
  - Supported in Java via **wait-sets** and **locks**

Guarding Via Suspension

- **Waiting for a condition to hold:**
  ```java
  synchronized (obj) {
      while (!condition) {
          try {
              obj.wait();
          } catch (InterruptedException ex) {
              ...
          }
      // make use of condition
  }
  ```

- **Changing a condition:**
  ```java
  synchronized (obj) {
      condition = true;
      obj.notifyAll(); // or obj.notify()
  }
  ```

- **Golden rule:** **Always** test a condition in a loop
  - Change of state may not be what you need
  - Condition may have changed again
    - No built-in protection from **‘barging’**
  - **Break** the rule **only** after you have **proven** it is safe to do so
Wait-sets and Notification

- Every Java Object has a wait-set
  - Can only be manipulated while the object lock is held
    - Otherwise IllegalMonitorStateException is thrown
- Threads enter the wait-set by invoking \texttt{wait()}
  - \texttt{wait()} atomically releases the lock and suspends the thread
    - Including a lock held multiple times—makes the object ‘open’
    - No other held locks are released
  - Optional timed-wait: \texttt{wait(long millis)}
    - No direct indication that a time-out occurred
    - \texttt{wait()} is equivalent to \texttt{wait(0)}—means wait forever
    - Nanosecond version too

Wait-sets and Notification (cont …)

- Threads are released from the wait-set when:
  - \texttt{notifyAll()} is invoked on the object
    - All threads released
  - \texttt{notify()} is invoked on the object
    - One thread selected at ‘random’ for release
  - The specified time-out elapses
  - The thread has its \texttt{interrupt()} method invoked
    - \texttt{InterruptedException} thrown
  - A spurious wakeup occurs
    - Not (yet!) spe\texttt{ç}’ed but an inherited property of underlying synchronization mechanisms eg. POSIX condition variables
- Lock is always reacquired before \texttt{wait()} returns
  - Lock count is restored
  - Can’t be acquired until notifying thread releases it
  - Released thread contends with all other threads for the lock
Wait-sets and Notifications (cont...)

- Consider `notify()` as an optimization which can only be used
  - When only one thread can benefit from the change of state, and
  - All threads are waiting for the same change of state
    - Or else another `notify()` is done by the released thread
  - And these conditions will also hold in all subclasses
- Conditional notification is another optimization
  - When you know what state changes are being waited upon
    - Subclasses may invalidate your ‘knowledge’
- Use of `wait()`, `notifyAll()` and `notify()` similar to
  - Condition queues of classic Monitors
  - Condition variables of POSIX PThreads API
  - But only one ‘queue’ per object
    - Great complicates some designs and easily leads to ‘nested monitor lockouts’
- Any Java object can be used just for its wait-set and/or lock

Example: Guarded Bounded Buffer

```java
public class GuardedBoundedBuffer implements Buffer {
    private List data;
    private final int capacity;

    public GuardedBoundedBuffer(int capacity) {
        data = new ArrayList(capacity);
        this.capacity = capacity;
    }

    public synchronized Object take() throws Failure {
        while (data.size() == 0)
            try {
                wait();
            } catch (InterruptedException ex) {throw new Failure();}
        Object temp = data.get(0);
        data.remove(0);
        notifyAll();
        return temp;
    }

    public synchronized void put(Object obj) throws Failure {
        while (data.size() == capacity)
            try {
                wait();
            } catch (InterruptedException ex) {throw new Failure();}
        data.add(obj);
        notifyAll();
    }

    public synchronized int size() { return data.size(); }
    public int capacity() { return capacity; }
}
```
Timeouts

- Intermediate points between balking and guarding
  - Can vary timeout parameter from zero to infinity
- Useful for heuristic detection of failures
  - Deadlocks, crashes I/O problems, network disconnects
- But cannot be used for high-precision timing or deadlines
  - Time can elapse between wait and thread resumption
  - Time can elapse after checking the time!
- Java implementation constraints
  - `wait(ms)` does not automatically tell you if it returns because of notification vs timeout
  - Must check for both. Order and style of checking can matter, depending on
    - If always OK to proceed when condition holds
    - If timeouts signify errors
    - No way to establish with 100% certainty that timeout occurred

Timeout Example

```java
public synchronized void put(Object obj, long timeout) throws Failure {
    if (timeout <= 0) // disallowing zero avoids semantic problems
        throw new IllegalArgumentException("timeout must be > 0");

    long timeleft = timeout;
    long start = System.currentTimeMillis();

    while (data.size() == capacity) {
        try { wait(timeleft); }
            catch (InterruptedException ex) { throw new Failure(); }

        // notified, timed-out or spurious?
        if (data.size() < capacity)
            break; // condition holds - don't care if we timed out
        else { // maybe a timeout
            long elapsed = System.currentTimeMillis() - start;
            timeleft = timeout - elapsed;
            if (timeleft <= 0) throw new Failure("Timed-out");
        } // spurious so wait again

    data.add(obj);
    notifyAll();
}
```
### Containment and Monitor Methods

class Part {
    protected boolean cond = false;
    synchronized void await() {
        while (!cond)
            try { wait(); }
            catch(InterruptedException ex) { ... }
    }
    synchronized void signal(boolean c) {
        cond = c; notifyAll();
    }
}

class Whole{
    final Part part = new Part();
    synchronized void rely() { part.await(); }
    synchronized void set(boolean c){
        part.signal(c);
    }
}

- What happens when Whole.rely() is called?

### Nested Monitors

- If thread T calls Whole.rely
  - It waits within part
  - The lock to Whole is retained while T is suspended
  - No other thread will ever unblock it via Whole.set
    - Nested Monitor Lockout
- Policy clash between guarding by Part and containment by Whole
  - One or the other should be changed
Avoiding Nested Monitors

- Adapt internal containment locking pattern

  ```java
class Whole {
    class Part {
      public void await() {
        synchronized (Whole.this) {
          while (...) Whole.this.wait();
        }
      }
    }
  }
  
  Owner object provides lock and wait-set
- Invert locking order so that outer lock is released before wait
  Requires special steps to maintain atomicity
- Create special condition objects eg. Semaphores, Events
  Condition methods are never invoked while holding locks

Optimistic Policies: Trying

- Isolate state into versions
  E.g. by grouping into a helper class
- Isolate state changes to atomic commit method that swaps in new state
- On method entry
  Save/record current state
  Apply action to new state
- Only commit if
  Action succeeds and current state version is unchanged
- If can’t commit: fail or retry
  Failures are clean (no side effects)
  Retry policy is variation of a busy-wait
- Only applicable if actions fully reversible
  No I/O or thread construction unless safely cancellable
  All internally called methods must be undoable
Optimistic Techniques

- Variations for recording versions of mutable data:
  - Immutable helper classes
  - Version numbers
  - Transaction ids
  - Time-stamps
- May be more efficient than guarded waits when:
  - Conflicts are rare and when running on multiple CPUs
- Retries can livelock unless proven wait-free
  - Analog of deadlock in guarded waits
  - Should arrange to fail after a certain time or number of attempts

---

Optimistic Bounded Counter

```java
public class OptimisticBoundedCounter {
    private final long MIN, MAX;
    private Long count; // MIN <= count <= MAX

    public OptimisticBoundedCounter(long min, long max) {
        MIN = min; MAX = max;
        count = new Long(MIN);
    }

    public long value() { return count().longValue(); }
    public synchronized Long count() { return count; }

    private synchronized boolean commit(Long oldc, Long newc) {
        boolean success = (count == oldc);
        if (success) count = newc;
        return success;
    }

    public void inc() throws InterruptedException {
        for (;;) {
            // retry-based
            if (Thread.interrupted())
                throw new InterruptedException();
            Long c = count(); // record current state
            long v = c.longValue();
            if (v < MAX && commit(c, new Long(v+1)))
                break;
            Thread.yield(); // a good idea in spin loops
        }
    }

    public void dec() { /* symmetrical */}
}
```

© 2003 David Holmes and Doug Lea
Specifying Policies

- Some policies are per-type
  - Optimistic approaches require all methods to conform
- Some policies can be specified per-call
  - Balking vs. Guarding vs. Guarding with time-out
- Options for specifying per-call policy:
  - Extra parameters
    
    ```java
    void put(Object x, long timeout )
    void put(Object x, boolean balk )
    ```
  - Different name for balking or guarding
    ```java
    boolean tryPut( Object x ) // balking
    void put( Object x ) // guarding
    ```
  - May need different exception signatures

Oneway Messages

- Conceptually one-way messages are sent with
  - No need for replies
  - No concern about failure (exceptions)
  - No dependence on termination of called method
  - No dependence on order that messages are received
  - But may sometimes want to cancel messages or resulting activities
- Once one-way message has been sent, `host` is ready to `accept` the next message

© 2003 David Holmes and Doug Lea
Thread Patterns for Oneway Messages

Thread-per-Message

client → host → handler

new thread

Thread-per-Object via Worker Threads or Pools

client → host → channel → handler

worker thread

Threads-Per-Message Web Server

Return to one-shot version of `startServer` but pass each accepted connection to a new thread for processing:

```java
// WebServer14.java
Thread serverThread;

public synchronized void startServer() throws ...
{
    if (serverThread != null)
        throw new IllegalStateException("Already started");
    serverThread = new Thread(new ConnectionHandler());
    serverThread.start();
}

private class ConnectionHandler implements Runnable {
    public void run() {
        // ...
        try {
            while (!Thread.interrupted()) {
                RequestHandler r =
                    new RequestHandler(server.accept());
                new Thread(r, "worker-thread").start();
            }
        }
        catch (InterruptedException ex) { /* ignore */ }
        catch (IOException ex) { /* report */ }
    }
}
```
Thread-Per-Object via Worker Threads

- Establish a producer-consumer chain
  - Producer
    - Reactive method just places message in a channel
      - Channel might be a buffer, queue, stream, etc
      - Message might be a Runnable command, event, etc
  - Consumer
    - Host contains an autonomous loop thread of form:
      ```java
      while (!Thread.interrupted()) {
        m = channel.take();
        process(m);
      }
      ```

- Common variants
  - Pools
    - Use more than one worker thread
  - Listeners
    - Separate producer and consumer in different objects

Web Server Using Worker Thread

```java
public interface Channel {
  // buffer, queue, stream etc
  Object take() throws InterruptedException;
  void put(Object obj) throws InterruptedException;
  int size();
}

// WebServer15.java
private Channel channel = new BoundedBuffer(); // synchronized
private class ConnectionHandler implements Runnable {
  public void run() {
    RequestHandler r = null;
    try {
      while (!Thread.interrupted()) {
        r = new RequestHandler(server.accept());
        channel.put(r);
      }
    } // ... interrupt and exception handling - more complex
  }
}
private class ChannelConsumer extends Thread {
  // Exception handling elided for simplicity
  // Also for simplicity, assumes channel has only one consumer
  public void run() {
    boolean stopProcessing = Thread.interrupted();
    while (!stopProcessing || channel.size() > 0) {
      ((Runnable) channel.take()).run();
      stopProcessing = Thread.interrupted();
    }
  }
}
```

© 2003 David Holmes and Doug Lea