Chapter: Containment and Monitor Methods

```java
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);
    }
}
```

Chapter: 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

Chapter: 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

Chapter: 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

Chapter: Optimistic Bounded Counter

```java
public class OptimisticBoundedCounter {
    private final long MAX, MIN;
    public OptimisticBoundedCounter(long min, long max) {
        MAX = max; MIN = min;
    }
    public boolean tryCommit(long c) {
        return c == count;
    }
    public synchronized long count() {
        return count;
    }
    public synchronized void inc() {
        count = count + 1;
    }
    public synchronized void dec() {
        count = count - 1;
    }
}
```
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
    - void put(Object x, long timeout)
    - void put(Object x, boolean balk)
  - Different name for balking or guarding
    - boolean tryPut(Object x) // balking
    - void put(Object x) // guarding
  - May need different exception signatures

Thread Creation Patterns

- Three general sets of patterns for introducing concurrency:
  - Autonomous loops
    - Establishing independent cyclic behaviour
  - Oneway messages
    - Sending messages without waiting for reply or termination
    - Improves availability of sender object
  - Interactive messages (not covered—see CPJ)
    - Requests that later result in reply or callback messages
    - Allows client to proceed concurrently for a while
- Most design ideas and semantics stem from active object models

Autonomous Loops

- Simple non-reactive active objects contain a run loop of form:
  ```java
  public void run() {
    while (!Thread.interrupted())
      doSomething();
  }
  ```
- Normally established with a constructor containing:
  ```java
  new Thread(this).start();
  ```
  - Or by a specific start method
  - Perhaps also setting priority and daemon status
  - Normally also support other methods called from other threads
  - Requires standard safety measures
- Common Applications
  - Animations, Simulations, Message buffer Consumers, Polling daemons that periodically sense state of world
  - This is the basic approach of our web server so far

Oneway Messages

- Conceptually oneway 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 oneway message has been sent, host is ready to accept the next message

Oneway Message Styles

- Some semantic choices
  - Asynchronous: Entire message send is independent
  - By far, most common style in reactive applications
  - Synchronous: Caller must wait until message is accepted
  - Basis for rendezvous protocols
  - Multicast: Message is sent to group of recipients
  - The group might not even have any members

Design Goals for Oneway Messages

- Safety
  - Local state changes should be atomic (normally, locked)
  - Typical need for locking leads to main differences vs single-threaded Event systems
- Safe guarding and failure policies, when applicable
- Availability
  - Minimize delay until host can accept another message
- Flow
  - The activity should progress with minimal contention
- Performance
  - Minimize overhead and resource usage
Thread Patterns for Oneway Messages

Thread-per-Message

client → host → handler

new thread

Thread-per-Object via Worker Threads or Pools

client → host → handler

worker thread

Thread Patterns for Oneway Messages

Thread-per-Message

client → host → handler

new thread

Thread-per-Object via Worker Threads or Pools

client → host → 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()
    {
        // ...
        RequestHandler r = new RequestHandler(server.accept());
        new Thread(r, "worker-thread").start();
    }
    // ... interrupt and exception handling - more complex
}
```

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()
    {
        // ...
        RequestHandler r = new RequestHandler(server.accept());
        new Thread(r, "worker-thread").start();
    }
    // ... interrupt and exception handling - more complex
}
```

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()
    {
        // ...
        RequestHandler r = new RequestHandler(server.accept());
        new Thread(r, "worker-thread").start();
    }
    // ... interrupt and exception handling - more complex
}
```

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 = new RequestHandler(server.accept());
        channel.put(r);
    }
    // ... interrupt and exception handling - more complex
}
private class ChannelConsumer extends Thread
{
    public void run()
    {
        boolean stopProcessing = Thread.interrupted();
        while (!stopProcessing || channel.size() > 0)
        {
            ((Runnable) channel.take()).run();
            if (!stopProcessing)
                stopProcessing = Thread.interrupted();
        }
    }
}
```

Channel Options

- Unbounded queues
  - Can exhaust resources if clients faster than handlers
- Bounded buffers
  - Can cause clients to block when full
- Leaky bounded buffers
  - For example, drop oldest if full
- Priority queues (run more important tasks first)
- Must ensure fairness
- Non-blocking channels
  - Must take evasive action if put or take fail or timeout

```
Thread Pools

Use a collection of worker threads, not just one

- Can limit maximum number and priorities of threads
- Dynamic worker thread management
- Sophisticated policy controls
- Often faster than thread-per-message for I/O bound actions

Thread Pools

Use a collection of worker threads, not just one

- Can limit maximum number and priorities of threads
- Dynamic worker thread management
- Sophisticated policy controls
- Often faster than thread-per-message for I/O bound actions

Thread Pools

Use a collection of worker threads, not just one

- Can limit maximum number and priorities of threads
- Dynamic worker thread management
- Sophisticated policy controls
- Often faster than thread-per-message for I/O bound actions

Thread Pools

Use a collection of worker threads, not just one

- Can limit maximum number and priorities of threads
- Dynamic worker thread management
- Sophisticated policy controls
- Often faster than thread-per-message for I/O bound actions

Thread Pools

Use a collection of worker threads, not just one

- Can limit maximum number and priorities of threads
- Dynamic worker thread management
- Sophisticated policy controls
- Often faster than thread-per-message for I/O bound actions
**Web Server Using Thread Pool**

```
import edu.oswego.cs.dl.util.concurrent.PooledExecutor;
private PooledExecutor pool; // WebServer16
poll = new PooledExecutor();
serverThread = new Thread(new ConnectionHandler());
// WebServer16

class ConnectionHandler implements Runnable {
public void run() {
RequestHandler r = new RequestHandler(server.accept());
pool.execute(r);
}
}
public void startServer() throws ...
{...

pool = new PooledExecutor();
serverThread = new Thread(new ConnectionHandler());
serverThread.start();
}
```

**Pools in Connection-Based Designs**

- For systems with many open connections (sockets), but relatively few active at any given time
- Multiplex the delegations to worker threads via polling
  - Requires underlying support for select/poll and nonblocking I/O
  - Supported in JDK1.4 `java.nio`

**Policies and Parameters for Thread Pools**

- The kind of channel used as task queue
  - Unbounded queue, bounded queue, synchronous hand-off, priority queue, ordering by task dependencies, stream, socket
- Bounding resources
  - Maximum number of threads
  - Minimum number of threads
  - “Warm” versus on-demand threads
  - “Warm” versus on-demand threads
- Saturation policy
  - Block, drop, producer-runs, etc
- These policies and parameters can interact in subtle ways!