Example: Web Server Statistics

- Track number of requests and average processing time
  - Statistics updated by server thread
  - Statistics reported via UI driven from the main application thread
- Shared data must be protected via synchronization
  - `synchronized` public accessor methods
  - `synchronized` statement within internal server method

```java
private long requests = 0;
private double avgTime = 0.0;

public synchronized long getRequests() {
    return requests;
}

public synchronized double getAverageTime() {
    return avgTime;
}
```

```java
private void processRequest(Socket sock) throws IOException {
    long start = System.currentTimeMillis();
    // .... Process
    long elapsed = System.currentTimeMillis() - start;
    synchronized(this) {
        avgTime = (avgTime*requests + elapsed)/(++requests);
    }
}
```

Safety requirements:
- Statistics are always valid (no storage conflicts)
- All statistics must be updated together
- Querying statistics should always yield a valid set of statistics

Functional requirements:
- Track number of requests
- Track average processing time per request
- Update each statistic at the end of each request
- Allow each statistic to be read
Isolating Dependent Representations

- Class `Webserver9` fails to satisfy all safety requirements
- No protection from interleaving such as:

<table>
<thead>
<tr>
<th>Main Thread</th>
<th>Server Thread</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>n = ws.getRequests();</code></td>
<td><code>...</code></td>
</tr>
<tr>
<td><code>...</code></td>
<td><code>processRequest(sock);</code></td>
</tr>
<tr>
<td><code>T = ws.getAverageTime();</code></td>
<td><code>...</code></td>
</tr>
</tbody>
</table>

◆ Main thread sees invalid set of statistics
- Old number of requests and new average processing time
- Avoid by splitting out dependent representation into a separate class

Split Representation Example

```java
public class WebServer10 {
    public static class Statistics {
        // immutable
        public final long requests;
        public final double avgTime;
        public Statistics(long requests, double avgTime) {
            this.requests = requests;
            this.avgTime = avgTime;
        }
    }
    private Statistics stats = new Statistics(0, 0.0);
    public synchronized Statistics getStatistics() {
        return stats;
    }
    // ...
    private void processRequest(Socket sock) throws IOException {
        // .... Process
        synchronized(this) {
            double total = stats.avgTime*stats.requests + elapsed;
            stats = new Statistics(stats.requests + 1,
                                   total / (stats.requests+1));
        }
    }
}
```

© 2003 David Holmes and Doug Lea
Split Objects

- Clients of principal classes can only view consistent states
- Splitting can be used in other contexts
  - Reducing lock contention
    - Associate a lock object with an isolable subset of state and functionality
  - Reducing “condition variable” contention
    - Associate a condition object with an isolable subset of methods with the same wait conditions
  - To be discussed in a later section
- Enabling rollback
  - Isolate some or all state in separate objects so you can save and track versions

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 cannot be shared
- Can be safely contained within a WebServer instance
  ```java
  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