

# Concurrency

## Chapters 2 & 3 (Goetz)

Listing 1.1 -----

```
import net.jcip.annotations.*;

@NotThreadSafe
public class UnsafeSequence {
    private int value;

    /**
     * Returns a unique value.
     */
    public int getNext() {
        return value++;
    }
}
```

Listing 1.2 -----

```
package net.jcip.examples;

import net.jcip.annotations.*;

@ThreadSafe
public class Sequence {
    @GuardedBy("this") private int nextValue;

    public synchronized int getNext() {
        return nextValue++;
    }
}
```

If multiple threads access the same mutable state variable without proper synchronization, your program is broken. There are three ways to fix it:

- Don't share state across threads
- Make the state variable immutable
- use synchronization

## Servlet and State

Listing 2.1 -----  
package net.jcip.examples;

```
import java.math.BigInteger;
import javax.servlet.*;
```

```
import net.jcip.annotations.*;
```

```
public class StatelessFactorizer extends
    GenericServlet implements Servlet {

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        encodeIntoResponse(resp, factors);
    }

    void encodeIntoResponse(ServletResponse resp,
        BigInteger[] factors) {

    }

    BigInteger extractFromRequest(ServletRequest req) {
        return new BigInteger("7");
    }

    BigInteger[] factor(BigInteger i) {
        // Doesn't really factor
        return new BigInteger[] { i };
    }
}
```

Listing 2.2 -----

```
package net.jcip.examples;
```

```
import java.math.BigInteger;
import javax.servlet.*;
```

```
import net.jcip.annotations.*;
```

```
@NotThreadSafe
public class UnsafeCountingFactorizer extends GenericServlet implements
Servlet {
    private long count = 0;

    public long getCount() {
        return count;
    }
}
```

```

public void service(ServletRequest req, ServletResponse resp) {
    BigInteger i = extractFromRequest(req);
    BigInteger[] factors = factor(i);
    ++count;
    encodeIntoResponse(resp, factors);
}

void encodeIntoResponse(ServletResponse res,
                        BigInteger[] factors) {
}

BigInteger extractFromRequest(ServletRequest req) {
    return new BigInteger("7");
}

BigInteger[] factor(BigInteger i) {
    // Doesn't really factor
    return new BigInteger[] { i };
}
}

```

the above is an example of **read-modify-write** = dangerous  
the program may produce an incorrect answer due to a **race condition**

## Race Condition in Lazy Initialization

```

package net.jcip.examples;

import net.jcip.annotations.*;

@NotThreadSafe
public class LazyInitRace {
    private ExpensiveObject instance = null;

    public ExpensiveObject getInstance() {
        if (instance == null)
            instance = new ExpensiveObject();
        return instance;
    }
}

class ExpensiveObject { }

```

## Atomic Long (java 5)

AtomicLong can solve problems associated with check-then-act operations that need to be atomic. java.util.concurrent.atomic package supports this and other classes.

Constructor Summary	
<a href="#">AtomicLong()</a>	Create a new AtomicLong with initial value 0.
<a href="#">AtomicLong(long initialValue)</a>	Create a new AtomicLong with the given initial value.

Method Summary	
long	<a href="#">addAndGet(long delta)</a> Atomically add the given value to current value.
boolean	<a href="#">compareAndSet(long expect, long update)</a> Atomically set the value to the given updated value if the current value == the expected value.
long	<a href="#">decrementAndGet()</a> Atomically decrement by one the current value.
double	<a href="#">doubleValue()</a> Returns the value of the specified number as a double.
float	<a href="#">floatValue()</a> Returns the value of the specified number as a float.
long	<a href="#">get()</a> Get the current value.
long	<a href="#">getAndAdd(long delta)</a> Atomically add the given value to current value.
long	<a href="#">getAndDecrement()</a> Atomically decrement by one the current value.
long	<a href="#">getAndIncrement()</a> Atomically increment by one the current value.
long	<a href="#">getAndSet(long newValue)</a>

	Set to the give value and return the old value.
long	<a href="#"><u>incrementAndGet()</u></a> Atomically increment by one the current value.
int	<a href="#"><u>intValue()</u></a> Returns the value of the specified number as an int.
long	<a href="#"><u>longValue()</u></a> Returns the value of the specified number as a long.
void	<a href="#"><u>set(long newValue)</u></a> Set to the given value.
<a href="#"><u>String</u></a>	<a href="#"><u>toString()</u></a> Returns the String representation of the current value.
boolean	<a href="#"><u>weakCompareAndSet(long expect, long update)</u></a> Atomically set the value to the given updated value if the current value == the expected value.

```

package net.jcip.examples;

import java.math.BigInteger;
import java.util.concurrent.atomic.*;
import javax.servlet.*;

import net.jcip.annotations.*;

/**
 * CountingFactorizer
 *
 * Servlet that counts requests using AtomicLong
 *
 */
@ThreadSafe
public class CountingFactorizer extends GenericServlet implements Servlet {
    private final AtomicLong count = new AtomicLong(0);

    public long getCount() { return count.get(); }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = factor(i);
        count.incrementAndGet(); // *** atomic operation
        encodeIntoResponse(resp, factors);
    }

    void encodeIntoResponse(ServletResponse res,
                           BigInteger[] factors) {}

```

```

    BigInteger extractFromRequest(ServletRequest req) {return null; }

    BigInteger[] factor(BigInteger i) { return null; }
}

```

When practical use existing thread-safe objects. like AtomicLong to manage class state.

## AtomicReference

### Constructor Summary

[AtomicReference](#)()

Create a new AtomicReference with null initial value.

[AtomicReference](#)([V](#) initialValue)

Create a new AtomicReference with the given initial value.

### Method Summary

boolean	<a href="#">compareAndSet</a> ( <a href="#">V</a> expect, <a href="#">V</a> update) Atomically set the value to the given updated value if the current value == the expected value.
<a href="#">V</a>	<a href="#">get</a> () Get the current value.
<a href="#">V</a>	<a href="#">getAndSet</a> ( <a href="#">V</a> newValue) Set to the given value and return the old value.
void	<a href="#">set</a> ( <a href="#">V</a> newValue) Set to the given value.
<a href="#">String</a>	<a href="#">toString</a> () Returns the String representation of the current value.
boolean	<a href="#">weakCompareAndSet</a> ( <a href="#">V</a> expect, <a href="#">V</a> update) Atomically set the value to the given updated value if the current value == the expected value.

```
package net.jcip.examples;
```

```

import java.math.BigInteger;
import java.util.concurrent.atomic.*;
import javax.servlet.*;

import net.jcip.annotations.*;

/**
 * UnsafeCachingFactorizer
 *
 * Servlet that attempts to cache its last result without adequate atomicity
 *
 */

@NotThreadSafe
public class UnsafeCachingFactorizer extends GenericServlet implements
Servlet {
    private final AtomicReference<BigInteger> lastNumber
        = new AtomicReference<BigInteger>();
    private final AtomicReference<BigInteger[]> lastFactors
        = new AtomicReference<BigInteger[]>();

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber.get()))
            encodeIntoResponse(resp, lastFactors.get());
        else {
            BigInteger[] factors = factor(i);
            lastNumber.set(i);
            lastFactors.set(factors);
            encodeIntoResponse(resp, factors);
        }
    }

    void encodeIntoResponse(ServletResponse resp, BigInteger[] factors) {
    }

    BigInteger extractFromRequest(ServletRequest req) {
        return new BigInteger("7");
    }

    BigInteger[] factor(BigInteger i) {
        // Doesn't really factor
        return new BigInteger[]{i};
    }
}

```

Problem: Despite use of **AtomicReference**, there are possible race conditions.

- There is an invariant here such that the factors cached in **lastFactors** equal the value cached in **lastNumber**. The servlet is only correct if this invariant always holds.
- When multiple variables participate in an invariant, they are not independent. Thus when updating one you must update the other atomically

Solution 1:

```
package net.jcip.examples;

import java.math.BigInteger;
import javax.servlet.*;

import net.jcip.annotations.*;

/**
 * SynchronizedFactorizer
 *
 * Servlet that caches last result
 */

@ThreadSafe
public class SynchronizedFactorizer extends GenericServlet implements Servlet
{
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;

    public synchronized void service(ServletRequest req,
                                     ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        if (i.equals(lastNumber))
            encodeIntoResponse(resp, lastFactors);
        else {
            BigInteger[] factors = factor(i);
            lastNumber = i;
            lastFactors = factors;
            encodeIntoResponse(resp, factors);
        }
    }

    void encodeIntoResponse(ServletResponse resp,
                           BigInteger[] factors) {
    }

    BigInteger extractFromRequest(ServletRequest req) {
        return new BigInteger("7");
    }

    BigInteger[] factor(BigInteger i) {
        // Doesn't really factor
        return new BigInteger[] { i };
    }
}

```

Good??

## CachedFactorizer

```
package net.jcip.examples;

import java.math.BigInteger;
import javax.servlet.*;

import net.jcip.annotations.*;

/**
 * CachedFactorizer
 * <p/>
 * Servlet that caches its last request and result
 *
 */
@ThreadSafe
public class CachedFactorizer extends GenericServlet implements Servlet {
    @GuardedBy("this") private BigInteger lastNumber;
    @GuardedBy("this") private BigInteger[] lastFactors;
    @GuardedBy("this") private long hits;
    @GuardedBy("this") private long cacheHits;

    public synchronized long getHits() {
        return hits;
    }

    public synchronized double getCacheHitRatio() {
        return (double) cacheHits / (double) hits;
    }

    public void service(ServletRequest req, ServletResponse resp) {
        BigInteger i = extractFromRequest(req);
        BigInteger[] factors = null;
        synchronized (this) {
            ++hits;
            if (i.equals(lastNumber)) {
                ++cacheHits;
                factors = lastFactors.clone();
            }
        }
        if (factors == null) {
            factors = factor(i);
            synchronized (this) {
                lastNumber = i;
                lastFactors = factors.clone();
            }
        }
        encodeIntoResponse(resp, factors);
    }

    void encodeIntoResponse(ServletResponse resp,
                           BigInteger[] factors) {
```

```

    }

    BigInteger extractFromRequest(ServletRequest req) {
        return new BigInteger("7");
    }

    BigInteger[] factor(BigInteger i) {
        // Doesn't really factor
        return new BigInteger[]{i};
    }
}

```

## Chapter 3.

### Example 3.1

```

package net.jcip.examples;

/**
 * NoVisibility
 * <p/>
 * Sharing variables without synchronization
 *
 */

public class NoVisibility {
    private static boolean ready;
    private static int number;

    private static class ReaderThread extends Thread {
        public void run() {
            while (!ready)
                Thread.yield();
            System.out.println(number);
        }
    }

    public static void main(String[] args) {
        new ReaderThread().start();
        number = 42;
        ready = true;
    }
}

```

Is this a problem??  
 What are the possible outputs?

## The Stale Data Problem

It is always dangerous to share data without synchronization. The key issue here is that the compiler can reorder operations relative to a single thread view. Plus, there is no guarantee that another thread will see the most recent value if an older value is in the cache!

- When a thread reads a variable without synchronization, it may see a stale value but it will see a value written by another thread. This is called out-of-thin-air safety.
- out-of-thin-air safety applies to all variables except double and long that are not declared volatile.

For long and double you must use volatile or guard them with a lock.

## Volatile

- Compiler is put on notice that other threads may access them.
- Volatile vars are not cached or reordered.

```
package net.jcip.examples;

/**
 * CountingSheep
 * <p/>
 * Counting sheep
 */
public class CountingSheep {
    volatile boolean asleep;

    void tryToSleep() {
        while (!asleep)
            countSomeSheep();
    }

    void countSomeSheep() {
        // One, two, three...
    }
}
```

## Publication and Escape

Publication means making an internal state object available outside the scope of the containing class. This can occur by:

- storing it where other code can access it
- returning it from a non-private method
- passing it to another class

When this happens we say the object has escaped.

### Escape 1

```
package net.jcip.examples;

import java.util.*;

/**
 * Secrets
 *
 * Publishing an object
 **/

class Secrets {
    public static Set<Secret> knownSecrets;

    public void initialize() {
        knownSecrets = new HashSet<Secret>();
    }
}

class Secret {
}
```

### Escape 2

```
package net.jcip.examples;

/**
 * UnsafeStates
 * Allowing internal mutable state to escape
 *
 **/
```

```

class UnsafeStates {
    private String[] states = new String[]{
        "AK", "AL" /*...*/
    };

    public String[] getStates() {
        return states;
    }
}

```

## Immutability

An immutable object is one whose state cannot change. Immutable objects are always thread safe.

An object is immutable if:

1. It's state cannot be changed after construction
2. All fields are `final`, and
3. It is properly constructed - the `this` reference does not escape during construction

A **final variable** can only be assigned once. This assignment does not grant the variable [immutable](#) status. If the variable is a field of a class, it must be assigned in the constructor of its class. (Note: If the variable is a reference, this means that the variable cannot be re-bound to reference another object. But the object that it references is still mutable, if it was originally mutable.)

Listing 3.11

```

package net.jcip.examples;

import java.util.*;

import net.jcip.annotations.*;

/**
 * ThreeStooges
 * Immutable class built out of mutable underlying objects,
 * demonstration of candidate for lock elision
 */
@Immutable
public final class ThreeStooges {
    private final Set<String> stooges = new HashSet<String>();

    public ThreeStooges() {
        stooges.add("Moe");
        stooges.add("Larry");
        stooges.add("Curly");
    }

    public boolean isStooge(String name) {
        return stooges.contains(name);
    }

    public String getStoogeNames() {

```

```
List<String> stooges = new Vector<String>();
stooges.add("Moe");
stooges.add("Larry");
stooges.add("Curly");
return stooges.toString();
}
}
```