Java and Android Concurrency

Thread Safety

fausto.spoto@univr.it

git@bitbucket.org:spoto/java-and-android-concurrency.git

git@bitbucket.org:spoto/java-and-android-concurrency-examples.git

Object State

An object state is its data, stored in state variables such as its instance fields. It might include the state of other, dependent objects. It encoppasses any data that can affect its externally visible behavior

shared state: accessed by multiple threads

mutable state: it could change during its lifetime

Whenever more than one thread accesses a given state variable, and one of them might write to it, they all must coordinate their access to it using synchronization

State Encapsulation

When designing thread-safe classes, good object-oriented techniques – encapsulation, immutability, and clear specification of invariants – are your best friends

Your enemies

- public fields
- static fields
- mutability
- state leakage

It is far easier to design a class to be thread-safe than to retrofit it to thread-safety later

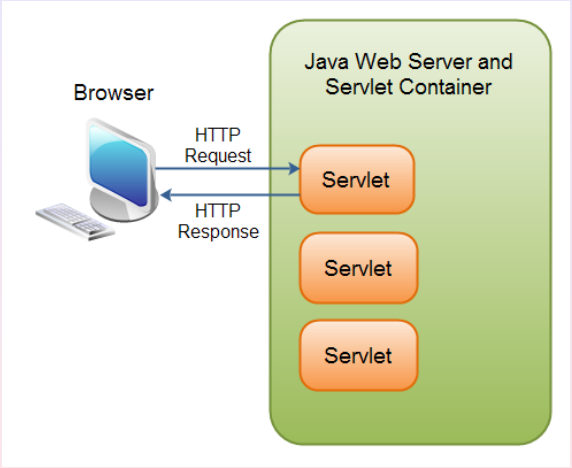
What is Thread Safety?

Largely philosophical question

A class is correct when it conforms to its specification. A class is thread-safe when it continues to behave correctly when accessed from multiple threads, regardless of the scheduling of those threads, and with no additional synchronization or other coordination on the part of the calling code

No set of operations – calls to public methods of reads or writes of public fields – performed sequentially or concurrently on instances of a thread-safe class can cause an instance to be in an invalid state

Java Servlets



Eclipse can create dynamic web projects and export them into .war files

Heroku: https://www.heroku.com

Create account at https://signup.heroku.com. Then install

```
sudo add-apt-repository "deb https://cli-assets.heroku.com/branches/stable/apt
./"
sudo apt install curl
curl -L https://cli-assets.heroku.com/apt/release.key | sudo apt-key add -
sudo apt-get update
sudo apt-get install heroku
```

`heroku --version`

heroku-cli/5.6.27-7c0098a (linux-amd64) go1.7.5

`heroku login`

Enter your Heroku credentials.
Email: fausto.spoto@univr.it
Password (typing will be hidden): verysafepassword
Authentication successful.

Heroku: Create and Deploy Application

heroku create

```
Creating app... done, limitless-bayou-56277
https://limitless-bayou-56277.herokuapp.com/
https://git.heroku.com/limitless-bayou-56277.git
```

Install the command line deployment plugin

```
heroku plugins:install heroku-cli-deploy
```

Deploy the application in Heroku

```
heroku war:deploy servlets.war --app limitless-bayou-56277
```

With a browser, go to `https://limitless-bayou-56277.herokuapp.com/StatelessFactorizer?number=250`

A Thread-Safe Factorizing Servlet

```
@ThreadSafe
@WebServlet("/StatelessFactorizer") // publication path
public class StatelessFactorizer extends HttpServlet {

  @Override
  protected void doGet(HttpServletRequest request, HttpServletResponse response) {
    doPost(request, response); // delegation
  }

  @Override
  protected void doPost(HttpServletRequest request, HttpServletResponse response) {
    BigInteger number = extractFromRequest(request);
    BigInteger[] factors = factor(number);  // see implementation in Eclipse
    encodeIntoResponse(response, factors);
  }

  protected BigInteger extractFromRequest(HttpServletRequest request) {
    return new BigInteger(request.getParameter("number"));
  }

  protected void encodeIntoResponse(HttpServletResponse response, BigInteger[] fs)
    response.getOutputStream().println(Arrays.toString(fs));
  }
}
```

Stateless Objects

There is only an instance of the servlet object

The servlet container receives many concurrent requests but creates a single instance of `it.univr.servlets.StatelessFactorizer`. All requests are routed to that instance, each running inside its own thread! There is no problem in this example, since the servlet keeps no state information in its fields

Stateless objects are always thread-safe

A Program that Connects to the Servlet

Before seeing other servlets, let us see how a client can connect to the `StatelessFactorizer` and ask its service:

```
public class ServletClient {
  public final static String SERVER
    = "https://limitless-bayou-56277.herokuapp.com/StatelessFactorizer?number=250";

  public static void main(String[] args)
      throws MalformedURLException, IOException {

    URL url = new URL(SERVER);
    URLConnection conn = url.openConnection();

    try (BufferedReader in = new BufferedReader(new InputStreamReader
        (conn.getInputStream()))) {

      String response;
      while ((response = in.readLine()) != null)
        System.out.println(response);
    }
  }
}
```

Let us Count the Number of Requests

```
@NotThreadSafe
@WebServlet("/UnsafeCountingFactorizer")
public class UnsafeCountingFactorizer extends StatelessFactorizer {
  private long count = 0;

  public long getCount() {
    return count;
  }

  @Override
  protected void doPost(HttpServletRequest request,
                        HttpServletResponse response) {
    BigInteger number = extractFromRequest(request);
    BigInteger[] factors = factor(number);
    ++count;
    encodeIntoResponse(response, factors);
  }
}
```

Non-Atomic Operations

The addition of just a bit of shared, mutable state makes the servlet non-thread-safe, because of a non-atomic operation `++count`. It gets compiled into many Java bytecode instructions:

```
aload 0
aload 0
getfield count:L       // read
const 1L
ladd                   // modify
putfield count:L       // write
```

Beware of

- read-modify-write operations on shared, mutable state
- check-then-act sequences on shared, mutable state

Race Condition

A race condition occurs when the correctness of a computation depends on the relative timing or interleaving of multiple threads by the runtime; in other words, when getting the right answer relies on lucky timing

A typical example of race condition is in the initialization of shared, meant to be unique instances of objects:

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

  public ExpensiveObject getInstance() {
    if (instance == null) // check
      instance = new ExpensiveObject(); // then act

    return instance;
  }
}
```

Atomicity

Operations *A* and *B* are atomic with respect to each other if, from the perspective of a thread executing *A*, when another thread executes *B*, either all of *B* has executed ot none of it has. An atomic operation is one that is atomic with respect to all operations, including itself, that operate on the same state

```
@ThreadSafe @WebServlet("/CountingFactorizer")
public class CountingFactorizer extends StatelessFactorizer {
  private final AtomicLong count = new AtomicLong(0L);

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

  @Override
  protected void doPost(HttpServletRequest request, HttpServletResponse response) {
    BigInteger number = extractFromRequest(request);
    BigInteger[] factors = factor(number);
    count.incrementAndGet();
    encodeIntoResponse(response, factors);
  }
}
```

Let us Cache the Last Request Result

```
@NotThreadSafe
@WebServlet("/UnsafeCachingFactorizer")
public class UnsafeCachingFactorizer extends StatelessFactorizer {
  private final AtomicReference<BigInteger> lastNumber = new AtomicReference<>();
  private final AtomicReference<BigInteger[]> lastFactors = new AtomicReference<>()

  @Override
  protected void doPost(HttpServletRequest request, HttpServletResponse response) {
    BigInteger number = extractFromRequest(request);
    if (number.equals(lastNumber.get()))
      encodeIntoResponse(response, lastFactors.get());
    else {
      BigInteger[] factors = factor(number);
      lastNumber.set(number);
      lastFactors.set(factors);
      encodeIntoResponse(response, factors);
    }
  }
}
```

From One State Variable to Two State Variables

Since a single state variable of type long can be used thread-safely by translating it into an `AtomicLong`, we could have expected to do the same with two state variables of reference type, by using `AtomicReference`

The result is not thread-safe!

There is an implicit link between the values of the two (thread-safe) state variables. Hence, to preserve state consistency, such related state variables must be updated in a single atomic operation

Recovering Thread-Safeness through Synchronization

In order to make updates to the two state variables atomic, they must be embedded inside the same `synchronized` block, hence exploiting the mutex nature of Java's intrinsic locks

```
@ThreadSafe @WebServlet("/SynchronizedFactorizer")
public class SynchronizedFactorizer extends StatelessFactorizer {
  private @GuardedBy("this") BigInteger lastNumber;
  private @GuardedBy("this") BigInteger[] lastFactors;

  @Override protected synchronized void doPost(...) {
    BigInteger number = extractFromRequest(request);
    if (number.equals(lastNumber)) encodeIntoResponse(response, lastFactors);
    else {
      BigInteger[] factors = factor(number); lastNumber = number;
      lastFactors = factors; encodeIntoResponse(response, factors);
    }
  }
}
```

We have recovered thread-safety at the price of efficiency: only one `SynchronizedFactorizer` can run at a time. This is not what concurrency was meant for

Reentrancy: A Natural Choice for an OO Language

Java's intrinsic locks are reentrant, that is, if a thread tries to acquire alock that it already holds, the request succeeds

This is necessary in an object-oriented language, or otherwise overriding of synchronized methods would deadlock:

```
public class Widget {
  public synchronized void doSomething() { ... }
}

public class LoggingWidget extends Widget {
  public synchronized void doSomething() {
    System.out.println(toString() + ": calling doSomething");
    super.doSomething();
  }
}
```

Guarding State with a Lock

For each mutable state variable that may be accessed by more than one thread, all accesses to that variables (both for writing and for reading) must be performed with the same lock held. In that case, we say that the variable is guarded by that lock

For every invariant that involves more than one variable, all the variables involved in that invariant must be guarded by the same lock

Make clear to maintainers which lock is used to access a shared, mutable variable. This is the goal of the `@GuardedBy` annotation

Can we save the world by making everything `synchronized` ? Not really...

```
if (!vector.contains(element))
  vector.add(element)
```

Performance

Making the whole servlet `doPost()` method `synchronized` restored thread-safety at the price of performance: only a thread can execute at a time. Let us try to put inside `synchronized` blocks only those portions of code that really need synchronization

Avoid holding locks during lengthy computations or operations at risk of not completing quickly such as network or console I/O (or sleep!)

A Factorizer that Keeps a Cache, with Good Performance

```
@ThreadSafe
@WebServlet("/CachedFactorizer")
public class CachedFactorizer extends StatelessFactorizer {
  private @GuardedBy("this") BigInteger lastNumber;
  private @GuardedBy("this") BigInteger[] lastFactors;
  private @GuardedBy("this") long hits;
  private @GuardedBy("this") long cacheHits;

  public synchronized long getHits() {
    return hits;
  }

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

A Factorizer that Keeps a Cache, with Good Performance

```
  @Override protected void doPost(...) {
    BigInteger number = extractFromRequest(request);
    BigInteger[] factors = null;

    synchronized (this) {
      ++hits;
      if (number.equals(lastNumber)) {
        ++cacheHits;
        factors = lastFactors;
      }
    }

    if (factors == null) {
      factors = factor(number); // long operation: outside synchronization!

      synchronized (this) {
        lastNumber = number;
        lastFactors = factors;
      }
    }

    encodeIntoResponse(response, factors);
  }
}
```

Exercise 1

Write a web application implementing a chat server, with two servlets:

Add a message to the chat

`AddMessage?author=AAAA&text=TTTT`

List the last messages of the chat

`ListMessages?howmany=HHHH`

If there are fewer messages, only lists those available. The list is provided in the output of the servlet, in increasing chronological order, as a sequence of XML messages:

```
<message>
  <author>
    Fausto
  </author>
  <text>
    Hello, are you listening?
  </text>
</message>
```

Exercise 1: Suggestion

Servlets have a context, holding data that must be made available to the whole application, that is, to all servlets of the same web application. This context can be accessed by writing:

```
ServletContext context = getServletContext()
```

Data can be stored and retrieved from the context by using its methods:

- `setAttribute(String key, Object value)`
- `Object getAttribute(String key)`, which yields `null` if the attribute is unknown

Since attribute values are shared across all instances of all servlets, they must be thread-safe

In the exercise, the list of chat messages might be an attribute

Exercise 2

Write a command-line client to the chat web application, that allows one to post new messages to the chat, by specifying author and text, and to list the last (up to) 10 messages in the chat, in increasing chronological order, such as:

```
Fausto says:
Hello, are you listening?

Linda says:
Kind of, I'm busy writing servlets

Fausto says:
Wow, you are a Java expert!
```

Share the same chat server across many clients and try its concurrent use