The Secrets of Concurrency

Dr Heinz M. Kabutz

www.javaspecialists.eu

Javaspecialists.eu
java training

Heinz Kabutz

- **Brief Biography**
  - **German from Cape Town, now lives in Chania on island of Crete**
  - **The Java Specialists' Newsletter**
    - **132 countries**
  - **Java Champion since 2005**
  - **JavaOne Rock Star 2012**


Join us here for an advanced Java course!
18-21 June 2013

2.2: The Secrets of Concurrency

- **Writing correct concurrent code can be a real challenge; only *perfect* is good enough**
- **You need to synchronize in the precisely correct places**
  - **Too much synchronization and you risk deadlock and contention**
  - **Too little synchronization and you risk seeing early writes, corrupt data, race conditions and stale local copies of fields**
- **In this section, we will look at ten laws that will make it easier for you to write correct thread-safe code.**

2.2: The Secrets of Concurrency

- **The ten laws that will help you write thread-safe code**
  - **Law 1: The Law of the Sabotaged Doorbell**
  - **Law 2: The Law of the Xerox Copier**
  - **Law 3: The Law of the Overstocked Haberdashery**
  - **Law 4: The Law of the Blind Spot**
  - **Law 5: The Law of the Leaked Memo**
  - **Law 6: The Law of the Corrupt Politician**
  - **Law 7: The Law of the Micromanager**
  - **Law 8: The Law of Cretan Driving**
  - **Law 9: The Law of Sudden Riches**
  - **Law 10: The Law of the Uneaten Lutefisk**

1. The Law of the Sabotaged Doorbell

Instead of arbitrarily suppressing interruptions, manage them better.

*** Removing the batteries from your doorbell to avoid hawkers also shuts out people that you want to have as visitors**

Law 1: The Law of the Sabotaged Doorbell

- Have you ever seen code like this?

```
try {
  Thread.sleep(1000);
} catch(InterruptedException ex) {
  // this won't happen here
}
```

- We will answer the following questions:
  - What does InterruptedException mean?
  - How should we handle it?

Shutting Down Threads

- **Shutdown threads when they are inactive**
  - **In WAITING or TIMED_WAITING states:**
    - `Thread.sleep()`
    - `BlockingQueue.get()`
    - `Semaphore.acquire()`
    - `wait()`
    - `join()`

Thread “interrupted” Status

- **You can interrupt a thread with:**
  - `someThread.interrupt();`
  - **Sets the “interrupted” status to** `true`
  - **What else?**
    - **If thread is in state WAITING or TIMED_WAITING, the thread immediately returns by throwing InterruptedException and sets “interrupted” status back to** `false`
    - **Else, the thread does nothing else. In this case,** `someThread.isInterrupted()` **will return** `true`

How to Handle InterruptedException?

- **Option 1: Simply re-throw InterruptedException**
  - **Approach used by java.util.concurrent**
  - **Not always possible if we are overriding a method**
- **Option 2: Catch it and return**
  - **Our current “interrupted” state should be set to true**
  - **Add a boolean volatile “running” field as backup mechanism**

```
while (running) {
  // do something
  try {
    TimeUnit.SECONDS.sleep(1);
  } catch (InterruptedException e) {
    Thread.currentThread().interrupt();
    break;
  }
}
```

Law 1: The Law of the Sabotaged Doorbell

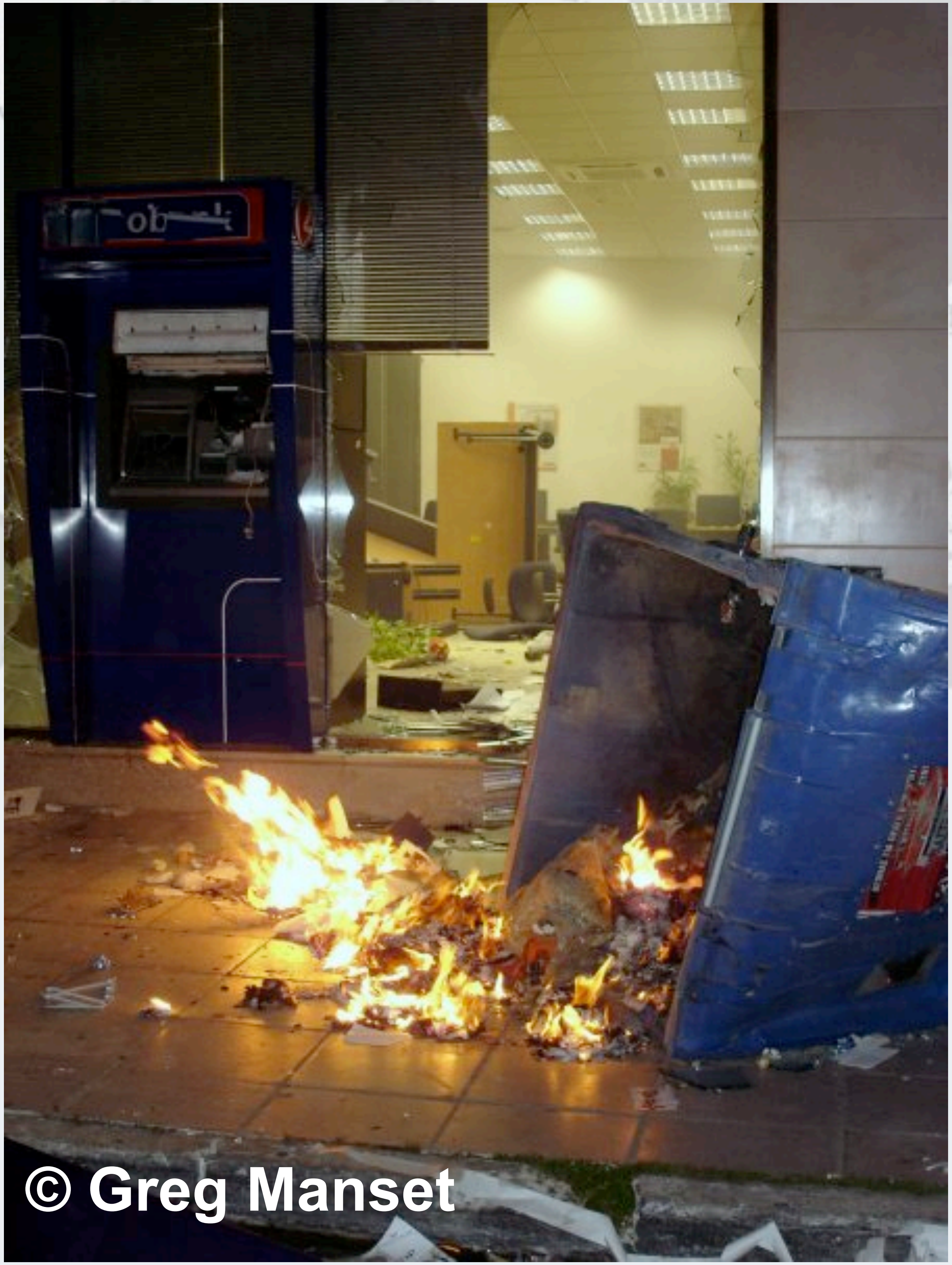
2. The Law of the Xerox Copier

Protect yourself by making copies of objects

*** Never give your originals to anyone, even a bank!**

"Safe as a Bank"

- **Our home loan application was lying on the desk the day this bank was trashed by rioters**
- ***Fortunately, we had only given them copies of our important documents!***



Law 2: The Law of the Xerox Copier

Law 2: The Law of the Xerox Copier

- **Immutable objects are always thread safe**
  - **No stale values, race conditions or early writes**
- **For concurrency, *immutable* means [Goetz'06]**
  - **State cannot be modified after construction**
  - **All the fields are final**
  - **'this' reference does not escape during construction**

How do we use an Immutable Object?

- **Whenever we want to change it, make a copy**
  - **e.g. String '+' operator produces a new String**
- **Additional GC expense, but concurrency is easier**

Basic Thread-Safe ArrayList

```
public class ImmutableArrayList<E> implements Iterable<E> {
  private final Object[] elements;

  public ImmutableArrayList() {
    this.elements = new Object[0];
  }

  private ImmutableArrayList(Object[] elements) {
    this.elements = elements;
  }

  public int size() { return elements.length; }

  public ImmutableArrayList<E> add(E o) {
    Object[] new_elements = new Object[elements.length + 1];
    System.arraycopy(elements, 0,
                     new_elements, 0, elements.length);
    new_elements[new_elements.length - 1] = o;
    return new ImmutableArrayList<E>(new_elements);
  }
```

Law 2: The Law of the Xerox Copier

Thread-Safe Iterator

```
  public Iterator<E> iterator() {
    return new Iterator<E>() {
      int pos = 0;

      public boolean hasNext() {
        return pos < elements.length;
      }

      public E next() {
        return (E) elements[pos++];
      }

      public void remove() {
        throw new UnsupportedOperationException();
      }
    };
  }
}
```

Law 2: The Law of the Xerox Copier

Using ImmutableArrayList

- **We use this in a more functional approach:**

```
ImmutableArrayList<String> ial =
  new ImmutableArrayList<String>();
ial = ial.add("Heinz").add("Max").add("Kabutz");
for (Object o : ial) {
  System.out.println("o = " + o);
}
```

3. The Law of the Overstocked Haberdashery

Having too many threads is bad for your application. Performance will degrade and debugging will become difficult.

*** Haberdashery: A shop selling sewing wares, e.g. threads and needles.**

Law 3: The Law of the Overstocked Haberdashery

- **Story: Client-side library running on server**
- **We will answer the following questions:**
  - **How many threads can you create?**
  - **What is the limiting factor?**
  - **How can we create more threads?**

Quick Demo

How many *inactive* threads can we create, before the JVM crashes?

Some JVMs Core Dump

```
Exception in thread "main" java.lang.OutOfMemoryError: unable
   to create new native thread
    at java.lang.Thread.start0(Native Method)
    at java.lang.Thread.start(Thread.java:597)
    at ThreadCreationTest$1.<init>(ThreadCreationTest:8)
    at ThreadCreationTest.main(ThreadCreationTest.java:7)
#
# An unexpected error has been detected by Java Runtime
   Environment:
#
#  Internal Error (455843455054494F4E530E4350500134) #
# Java VM: Java HotSpot(TM) Client VM (1.6.0_01-b06)
# An error report file with more information is saved as
   hs_err_pid22142.log
#
Aborted (core dumped)
```

How to Create More Threads?

- **We created about 2000 threads on Mac OS X**
  - **Had to kill with -9**
- **Stack size can cause OutOfMemoryError if too large**

Causing Thread Dumps

- **The jstack tool dumps threads of process**
  - Similar to CTRL+Break (Windows) or CTRL+\ (Unix)
- **For thread dump JSP page**
  - http://javaspecialists.eu/archive/Issue132.html
  - Sorted threads allow you to diff between calls

How Many Threads is Healthy?

- **Additional threads should improve performance**
- **Not too many active threads**
  - **± 4 active per core**
- **Inactive threads**
  - **Number is architecture specific**
    - **Consume memory**
    - **Can cause sudden death of the JVM**
    - **What if a few thousand threads suddenly become active?**

Law 3: The Law of the Overstocked Haberdashery

Traffic Calming

- Thread pooling good way to control number
- Use new ExecutorService
  - Fixed Thread Pool
- For small tasks, thread pools can be faster
  - Not main consideration
- See http://www.javaspecialists.eu/archive/Issue149.html

4. The Law of the Blind Spot

It is not always possible to
see what other threads (cars) are doing
with shared data (road)

Law 4: The Law of the Blind Spot

- **Java Memory Model allows thread to keep local copy of fields**
- **Your thread might not see another thread’s changes**
- **Usually happens when you try to avoid synchronization**

Calling shutdown() might have no effect

```
public class Runner {
  private boolean running = true;
  public void doJob() {
    while(running) {
      // do something
    }
  }
  public void shutdown() {
    running = false;
  }
}
```

Law 4: The Law of the Blind Spot

Why?

- **Thread1 calls** `doJob()` **and makes a local copy of** `running`
- **Thread2 calls** `shutdown()` **and modifies the value of field** `running`
- **Thread1 does not see the changed value of** `running` **and continues reading the local stale value**

Making Field Changes Visible

- **Three ways of preventing this**
  - **Make field volatile**
  - **Make field final puts a “freeze” on value**
  - **Make read and writes to field synchronized**
    - **Also includes new locks**

Better MyThread

```
public class Runner {
  private volatile boolean running = true;
  public void doJob() {
    while(running) {
      // do something
    }
  }
  public void shutdown() {
    running = false;
  }
}
```

Law 4: The Law of the Blind Spot

5. The Law of the Leaked Memo

The JVM is allowed to reorder your statements resulting in seemingly impossible states (seen from the outside)

*** Memo about hostile takeover bid left lying in photocopy machine**

Law 5: The Law of the Leaked Memo

- **If two threads call f() and g(), what are the possible values of a and b ?**

```
public class EarlyWrites {
  private int x;
  private int y;
  public void f() {
    int a = x;
    y = 3;
  }
  public void g() {
    int b = y;
    x = 4;
  }
}
```

Early writes can result in: a=4, b=3

The order of Things

- **Java Memory Model allows reordering of statements**
- **Includes writing of fields**
- **To the writing thread, statements appear in order**

How to Prevent This?

- **JVM is not allowed to move writes out of synchronized block**
  - **Allowed to move statements into a synchronized block**
- **Keyword** volatile **prevents early writes**
  - **From the Java Memory Model:**
    - **There is a happens-before edge from a write to a volatile variable v to all subsequent reads of v by any thread (where subsequent is defined according to the synchronization order)**

How to Cause This Error (Victor Grazi)

- **Actually this is very easy to produce. Here are the steps:**
  - **Write some production code that depends on correct ordering**
  - **Test it thoroughly**
    - **(Don't worry all tests will pass)**
  - **Release to production**
  - **Wait for the next corporate change freeze**
  - **Check your inbox - you will have 500 emails from irate users who were bitten by the bug**

6. The Law of the Corrupt Politician

In the absence of proper controls, corruption is unavoidable.

* **Lord Acton:** ***Power tends to corrupt. Absolute power corrupts absolutely.***

Law 6: The Law of the Corrupt Politician

- **Without controls, the best code can go bad**

```
public class BankAccount {
  private int balance;
  public BankAccount(int balance) {
    this.balance = balance;
  }
  public void deposit(int amount) {
    balance += amount;
  }
  public void withdraw(int amount) {
    deposit(-amount);
  }
  public int getBalance() { return balance; }
}
```

What happens?

- The += operation is not atomic
- Thread 1
  - Reads balance = 1000
  - Locally adds 100 = 1100
  - Before the balance written, Thread 1 is swapped out
- Thread 2
  - Reads balance=1000
  - Locally subtracts 100 = 900
  - Writes 900 to the balance field
- Thread 1
  - Writes 1100 to the balance field

Solutions

- **Pre Java 5**
  - **synchronized**
    - **But avoid using “this” as a monitor**
    - **Rather use a private final object field as a lock**
- **Java 5 and 6**
  - **Lock, ReadWriteLock**
  - **AtomicInteger – dealt with in The Law of the Micromanager**

Pre-Java 5

```
public class BankAccount {
  private int balance;
  private final Object lock = new Object();

  public BankAccount(int balance) {
    this.balance = balance;
  }

  public void deposit(int amount) {
    synchronized(lock) { balance += amount; }
  }
  public void withdraw(int amount) {
    deposit(-amount);
  }
  public int getBalance() {
    synchronized(lock) { return balance; }
  }
}
```

Law 6: The Law of the Corrupt Politician

ReentrantLocks

- Basic monitors cannot be interrupted and will never give up trying to get locked
  - The Law of the Uneaten Lutefisk
- Java 5 Locks can be interrupted or time out after some time
- Remember to unlock in a finally block

```
private final Lock lock =
  new ReentrantLock();

public void deposit(int amount) {
  lock.lock();
  try {
    balance += amount;
  } finally {
    lock.unlock();
  }
}

public int getBalance() {
  lock.lock();
  try {
    return balance;
  } finally {
    lock.unlock();
  }
}
```

Law 6: The Law of the Corrupt Politician

ReadWriteLocks

- Can distinguish read and write locks
- Use ReentrantReadWriteLock
- Then lock either the write or the read action
  - lock.writeLock().lock();
  - lock.writeLock().unlock();
- Careful: Starvation can happen!

```
private final ReadWriteLock lock =
  new ReentrantReadWriteLock();

public void deposit(int amount) {
  lock.writeLock().lock();
  try {
    balance += amount;
  } finally {
    lock.writeLock().unlock();
  }
}

public int getBalance() {
  lock.readLock().lock();
  try {
    return balance;
  } finally {
    lock.readLock().unlock();
  }
}
```

Law 6: The Law of the Corrupt Politician

7. The Law of the Micromanager

Even in life, it wastes effort and frustrates the other *threads*.

* *mi·cro·man·age*: **to manage or control with excessive attention to minor details.**

Law 7: The Law of the Micromanager

- **Thread contention is difficult to spot**
- **Performance does not scale**
- **None of the usual suspects**
  - CPU
  - Disk
  - Network
  - Garbage collection
- **Points to thread contention**

Real Example – *Don't Do This!*

- **"How to add contention 101"**
  - `String WRITE_LOCK_OBJECT = "WRITE_LOCK_OBJECT";`
- **Later on in the class**
  - `synchronized(WRITE_LOCK_OBJECT) { ... }`
- **Constant Strings are flyweights!**
  - **Multiple parts of code locking on one object**
  - **Can also cause deadlocks and livelocks**

AtomicInteger

- Thread safe without explicit locking
- Tries to update the value repeatedly until success
  - AtomicInteger.equals() is not overridden

```
public final int addAndGet(int delta) {
  for (;;) {
    int current = get();
    int next = current + delta;
    if (compareAndSet(current, next))
      return next;
    }
  }
}
```

Law 7: The Law of the Micromanager

```
import java.util.concurrent.atomic.AtomicInteger;

public class BankAccount {
  private final AtomicInteger balance =
    new AtomicInteger();

  public BankAccount(int balance) {
    this.balance.set(balance);
  }
  public void deposit(int amount) {
    balance.addAndGet(amount);
  }
  public void withdraw(int amount) {
    deposit(-amount);
  }
  public int getBalance() {
    return balance.intValue();
  }
}
```

Law 7: The Law of the Micromanager

8. The Law of Cretan Driving

The JVM does not enforce all the rules.
Your code is probably wrong, even if it works.

*** Don't *stop* at a stop sign if you treasure your car!**

Law 8: The Law of Cretan Driving

- **Learn the JVM Rules !**
- **Example from JSR 133 – Java Memory Model**
  - **VM implementers are encouraged to avoid splitting their 64-bit values where possible. Programmers are encouraged to declare shared 64-bit values as volatile or synchronize their programs correctly to avoid this.**

JSR 133 allows this – NOT a Bug

- **Method set() called by two threads with**
  - **0x12345678ABCD0000L**
  - **0x1111111111111111L**

```
public class LongFields {
  private long value;
  public void set(long v) { value = v; }
  public long get()       { return value; }
}
```

- **Besides obvious answers, “value” could now also be**
  - **0x11111111ABCD0000L or 0x1234567811111111L**

Law 8: The Law of Cretan Driving

Java Virtual Machine Specification

- Gives great freedom to JVM writers
- Makes it difficult to write 100% correct Java
  - It might work on all JVMs to date, but that does not mean it is correct!
- Theory vs Practice clash

Synchronize at the Right Places

- **Too much synchronization causes contention**
  - **As you increase CPUs, performance does not improve**
  - **The Law of the Micromanager**
- **Lack of synchronization leads to corrupt data**
  - **The Law of the Corrupt Politician**
- **Fields might be written early**
  - **The Law of the Leaked Memo**
- **Changes to shared fields might not be visible**
  - **The Law of the Blind Spot**

Law 8: The Law of Cretan Driving

9. The Law of Sudden Riches

Additional resources (faster CPU, disk or network, more memory) for seemingly stable system can make it unstable.

*** Sudden inheritance or lottery win …**

Law 9: The Law of Sudden Riches

- **Better hardware can break system**
  - **Old system: Dual processor**
  - **New system: Dual core, dual processor**

Faster Hardware

- **Latent defects show up more quickly**
  - **Instead of once a year, now once a week**
- **Faster hardware often coincides with higher utilization by customers**
  - **More contention**
- **E.g. DOM tree becomes corrupted**
  - **Detected problem by synchronizing all subsystem access**
  - **Fixed by copying the nodes whenever they were read**

10. The Law of the Uneaten Lutefisk

A deadlock in Java can only be resolved by restarting the Java Virtual Machine.

* Imagine a Viking father insisting that his stubborn child eat its lutefisk before going to bed

Law 10: The Law of the Uneaten Lutefisk

- **Part of program stops responding**
- **GUI does not repaint**
  - **Under Swing**
- **Users cannot log in anymore**
  - **Could also be The Law of the Corrupt Politician**
- **Two threads want what the other has**
  - **And are not willing to part with what they already have**

Using Multiple Locks

```
public class HappyLocker {
  private final Object lock = new Object();
  public synchronized void f() {
    synchronized(lock) {
      // do something ...
    }
  }
  public void g() {
    synchronized(lock) {
      f();
    }
  }
}
```

Law 10: The Law of the Uneaten Lutefisk

Finding the Deadlock

- **Pressing CTRL+Break or CTRL+\ or use jstack**

```
Full thread dump:
Found one Java-level deadlock:
=============================
"g()":
  waiting to lock monitor 0x0023e274 (object 0x22ac5808, a
   HappyLocker),
  which is held by "f()"
"f()":
  waiting to lock monitor 0x0023e294 (object 0x22ac5818, a
   java.lang.Object),
  which is held by "g()"
```

Deadlock Means You Are Dead ! ! !

- **Deadlock can be found with jconsole**
- **However, there is no way to resolve it**
- **Better to automatically raise critical error**
  - **Newsletter 130 – Deadlock Detection with new Lock**
  - **http://www.javaspecialists.eu/archive/Issue130.html**

Conclusion

- Threading is a lot easier when you know the rules
- Tons of free articles on JavaSpecialists.EU
  - http://www.javaspecialists.eu/archive
- Advanced Java Courses available
  - http://www.javaspecialists.eu/courses

Heinz Kabutz (heinz@kabutz.net)

- **Full presentation and more detail available on**
  - **kabutz.net/talks/confess13/secrets**
  - **Take a pen and write that down whilst we take questions**

The Secrets of Concurrency

heinz@kabutz.net

kabutz.net/talks/confess13/secrets

Javaspecialists.eu
Java training