'06 JavaPolis

11 - 15 DECEMBER ■ ANTWERP ■ BELGIUM

Adobe®

JBoss® a division of Red Hat

ORACLE®

Sun® microsystems

The Network is the Computer™

'06 JAVAPOLIS

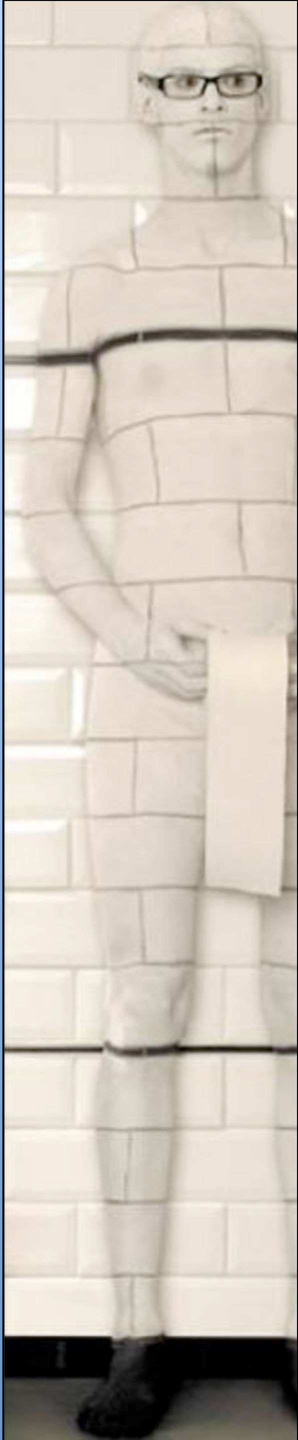
11 - 15 DECEMBER ■ ANTWERP ■ BELGIUM

Java Specialists in Action

Using dynamic proxies to write less code

Dr Heinz Kabutz
The Java Specialists' Newsletter

http://www.javaspecialists.eu



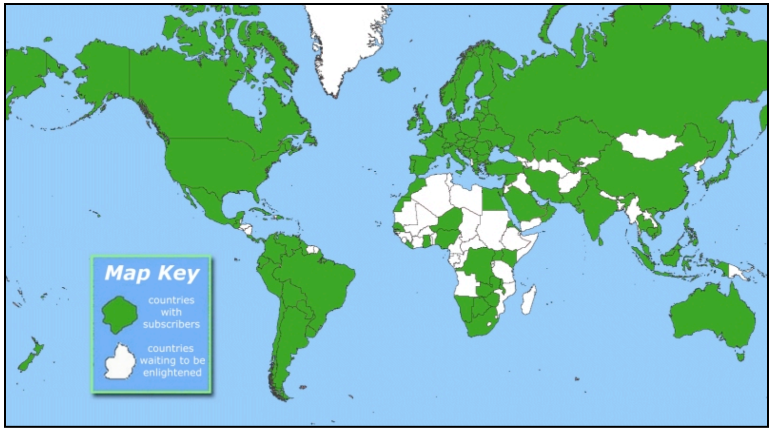
If you can read this, you're
sitting too close

Voyage of Discovery

A voyage of discovery through some of the more advanced topics in Java: dynamic proxies, references, generics and enums

Background – Who is Heinz?

- Author of The Java Specialists' Newsletter
  - 136 newsletters
  - Freely available
  - Over 30000 readers
  - **www.javaspecialists.eu**


Map Key
countries with subscribers
countries waiting to be enlightened

- Specialist Java trainer
  - Banks, insurance companies, telecoms, etc.
  - Intro to Java, Java 5 Delta, Java Patterns
- Living on an island in Greece – Crete

Introduction to Topic

- In this talk, we will look at:
  - Design Patterns
  - Dynamic Proxies in Java
  - Soft, Weak and Strong references
  - Some Java 5 features
- For additional free topics:
  - The Java™ Specialists' Newsletter
    - http://www.javaspecialists.eu
  - And find out how you can make `"hi there".equals("cheers!") == true`

Design Patterns

- Mainstream of OO landscape, offering us:
  - View into brains of OO experts
  - Quicker understanding of existing designs
    - e.g. Visitor pattern used by Annotation Processing Tool
  - Improved communication between developers
  - Readjust “thinking mistakes”

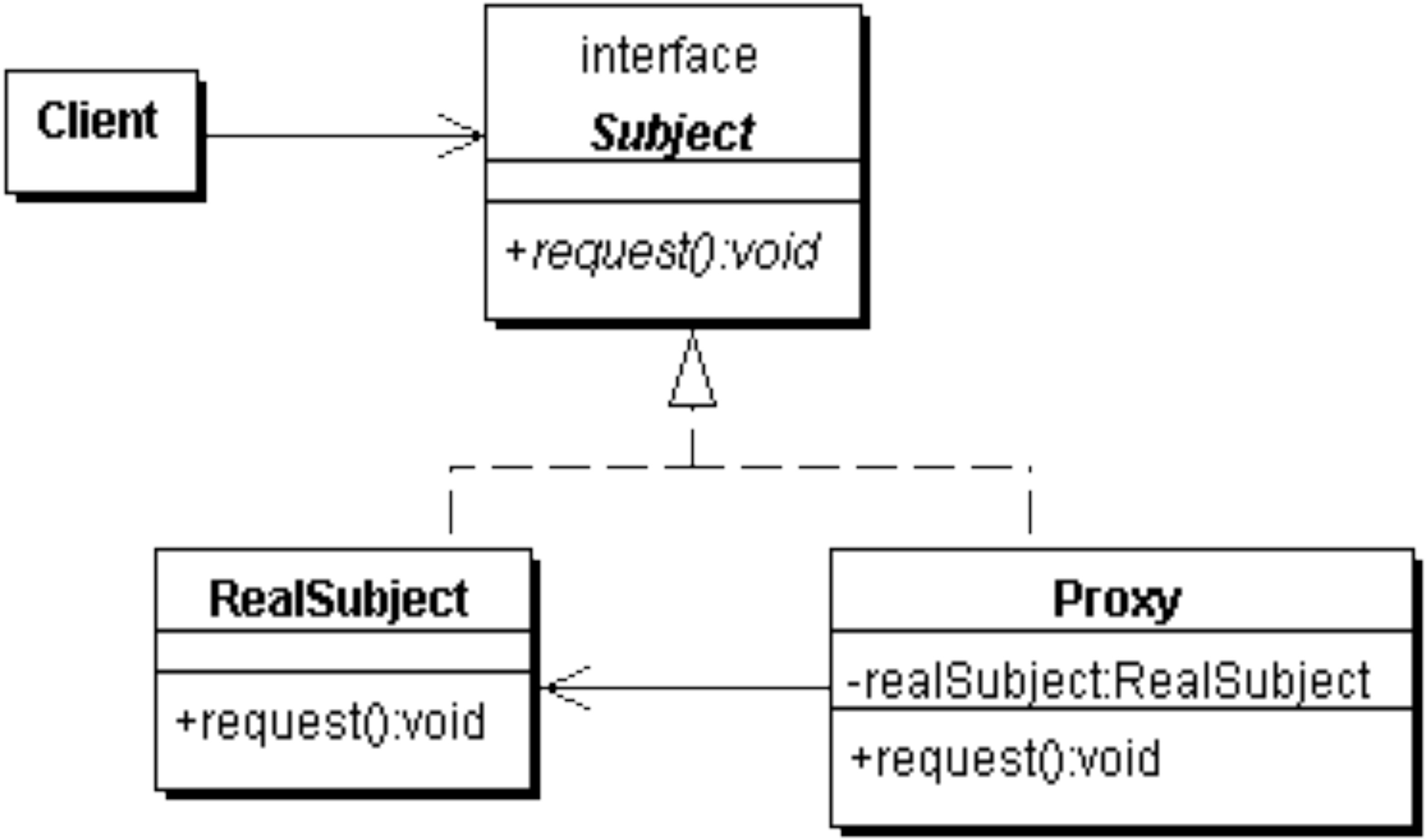
Vintage Wines

- Software Design is like good red wine
  - At first, quality of wine does not matter
    - As long as it has the right effect
  - With experience, you discern difference
  - As you become a connoisseur you experience the various textures you didn't notice before
    - Grown on the north slope in Italy on clay ground
- Warning: Once you are hooked, you will no longer be satisfied with inferior designs

Proxy Pattern

- Intent [GoF95]
  - Provide a surrogate or placeholder for another object to control access to it.

Proxy Structure



Types of Proxies in GoF

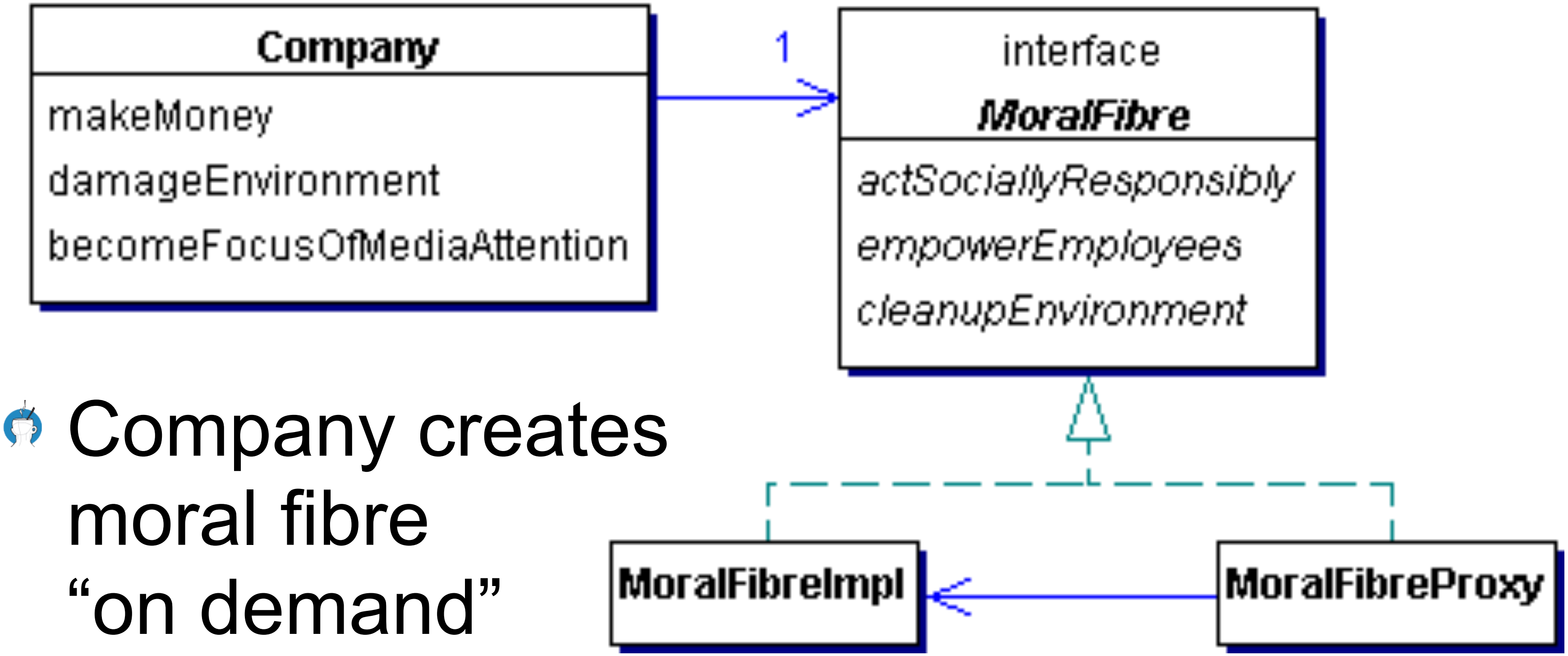
We will focus on this type

- Virtual Proxy
  - creates expensive objects on demand
- Remote Proxy
  - provides a local representation for an object in a different address space
- Protection Proxy
  - controls access to original object

Approaches to writing proxies

- Handcoded
  - Only for the very brave … or foolish
- Autogenerated code
  - RMI stubs and skeletons created by rmic
- Dynamic proxies
  - Available since JDK 1.3
  - Dynamically creates a new class at runtime
  - Flexible and easy to use

Model for example



- Company creates moral fibre “on demand”

```
public class Company {
  // set in constructor ...
  private final MoralFibre moralFibre;

  public void becomeFocusOfMediaAttention() {
    System.out.println("Look how good we are...");
    cash -= moralFibre.actSociallyResponsibly();
    cash -= moralFibre.cleanupEnvironment();
    cash -= moralFibre.empowerEmployees();
  }

  @Override
  public String toString() {
    Formatter formatter = new Formatter();
    formatter.format("%s has $ %.2f", name, cash);
    return formatter.toString();
  }
}
```

Quiz: Where is Autoboxing happening?

```
public interface MoralFibre {

  double actSociallyResponsibly();

  double empowerEmployees();

  double cleanupEnvironment();
}
```

Some parts of the code were left out to not flood you with too much information. Please contact me on **heinz@javaspecialists.eu** if you cannot get this baby to work.

```
public class MoralFibreImpl implements MoralFibre {
  // very expensive to create moral fibre!
  private byte[] costOfMoralFibre = new byte[900*1000];


  { System.out.println("Moral Fibre Created!"); }
  // AIDS orphans
  public double actSociallyResponsibly() {
    return costOfMoralFibre.length / 3;
  }
  // shares to employees
  public double empowerEmployees() {
    return costOfMoralFibre.length / 3;
  }
  // oiled sea birds
  public double cleanupEnvironment() {
    return costOfMoralFibre.length / 3;
  }
}
```

Handcoded Proxy

- Usually results in a lot of effort
- Shown just for illustration
- Good programmers have to be lazy
  - ➲ DRY principle
    - Don’t repeat yourself

```
public class MoralFibreProxy implements MoralFibre {
  private MoralFibreImpl realSubject;
  private MoralFibreImpl realSubject() {
    if (realSubject == null) { // need synchronization
      realSubject = new MoralFibreImpl();
    }
      return realSubject;
  }
  public double actSociallyResponsibly() {
    return realSubject().actSociallyResponsibly();
  }

  public double empowerEmployees() {
    return realSubject().empowerEmployees();
  }

  public double cleanupEnvironment() {
    return realSubject().cleanupEnvironment();
  }
}
```

```java
import static java.util.concurrent.TimeUnit.SECONDS;

public class WorldMarket0 {
  public static void main(String[] args)
      throws Exception {
    Company maxsol = new Company("Maximum Solutions",
        1000 * 1000, new MoralFibreProxy());
    SECONDS.sleep(2); // better than Thread.sleep();
    maxsol.makeMoney();
    System.out.println(maxsol);
    SECONDS.sleep(2);
    maxsol.damageEnvironment();
    System.out.println(maxsol);
    SECONDS.sleep(2);
    maxsol.becomeFocusOfMediaAttention();
    System.out.println(maxsol);
  }
}
```

```
Oh goodie!
Maximum Solutions has $ 2000000.00
Oops, sorry about that oilspill...
Maximum Solutions has $ 8000000.00
Look how good we are...
Moral Fibre Created!
Maximum Solutions has $ 7100000.00
```

Dynamic Proxies

- Handcoded proxy flawed
  - Previous approach broken – what if toString() is called?
  - Bugs would need to be fixed everywhere
- Dynamic Proxies
  - Allows you to write a method call handler
    - Invoked every time a method is called on interface
  - Easy to use

Defining a Dynamic Proxy

- We make a new instance of an interface class using java.lang.reflect.Proxy:

```
Object o = Proxy.newProxyInstance(
  Thread.currentThread().getContextClassLoader(),
  new Class[]{ interface to implement },
    implementation of InvocationHandler
);
```

- The result is an instance of ***interface to implement***
  - You could also implement several interfaces

```
import java.lang.reflect.*;

public class VirtualProxy implements InvocationHandler {
  private Object realSubject;
  private final Object[] constrParams;
  private final Constructor<?> subjectConstr;

  public VirtualProxy(Class<?> realSubjectClass,
                      Class[] constrParamTypes,
                      Object[] constrParams) {
    try {
      subjectConstr = realSubjectClass.
        getConstructor(constrParamTypes);
    } catch (NoSuchMethodException e) {
      throw new IllegalArgumentException(e);
    }
    this.constrParams = constrParams;
  }
```

Find constructor that matches given parameter types

Why did we not use varargs (…) for constrParamTypes and constrParams?

```
  private Object realSubject() throws Throwable {
    synchronized (this) {
      if (realSubject == null) {
        realSubject = subjectConstr.newInstance(
          constrParams);
      }
    }
    return realSubject;
  }
  public Object invoke(Object proxy, Method method,
                  Object[] args) throws Throwable {
    return method.invoke(realSubject(), args);
  }
}
```

- Whenever any method is invoked on the proxy object, it constructs real subject (if necessary) and delegates method call

A word about synchronization

- We need to **synchronize** whenever we check the value of the pointer
  - Otherwise several realSubject objects could be created
- We can synchronize on “this”
  - No one else will have a pointer to the object
- Double-checked locking broken pre-Java 5
  - It now works if you make the field **volatile**
  - Easier to get **synchronized** correct than **volatile**

Casting without Unchecked Warnings

- Cast to a specific class:
  - **subjIntf.cast(some_object)**
  - Allows you to do stupid things, like:

```
String name = String.class.cast(3);
```

Casting without Unchecked Warnings

- Cast a class to a typed class
  - With “forNamed” classes

```
Class<?> c = Class.forName( "some_class_name" );
Class<? extends SomeClass> c2 =
  c.asSubclass(SomeClass.class);
```

  - Allows you to do stupid things, like:

```
Class<?> c = Class.forName("java.lang.String");
Class<? extends Runnable> runner =
  c.asSubclass(Runnable.class);
Runnable r = runner.newInstance();
r.run();
```

Proxy Factory

- To simplify our client code, we define a Proxy Factory:
  - We want a return type of class **subjIntf**

```
import java.lang.reflect.*;

public class ProxyFactory {
  public static <T> T virtualProxy(Class<T> subjIntf,
           Class<? extends T> realSubjClass,
           Class[] constrParamTypes,
           Object[] constrParams) {
    return subjIntf.cast(Proxy.newProxyInstance(
       Thread.currentThread().getContextClassLoader(),
       new Class[] {subjIntf},
       new VirtualProxy<T>(realSubjClass,
           constrParamTypes, constrParams)));
  }
```

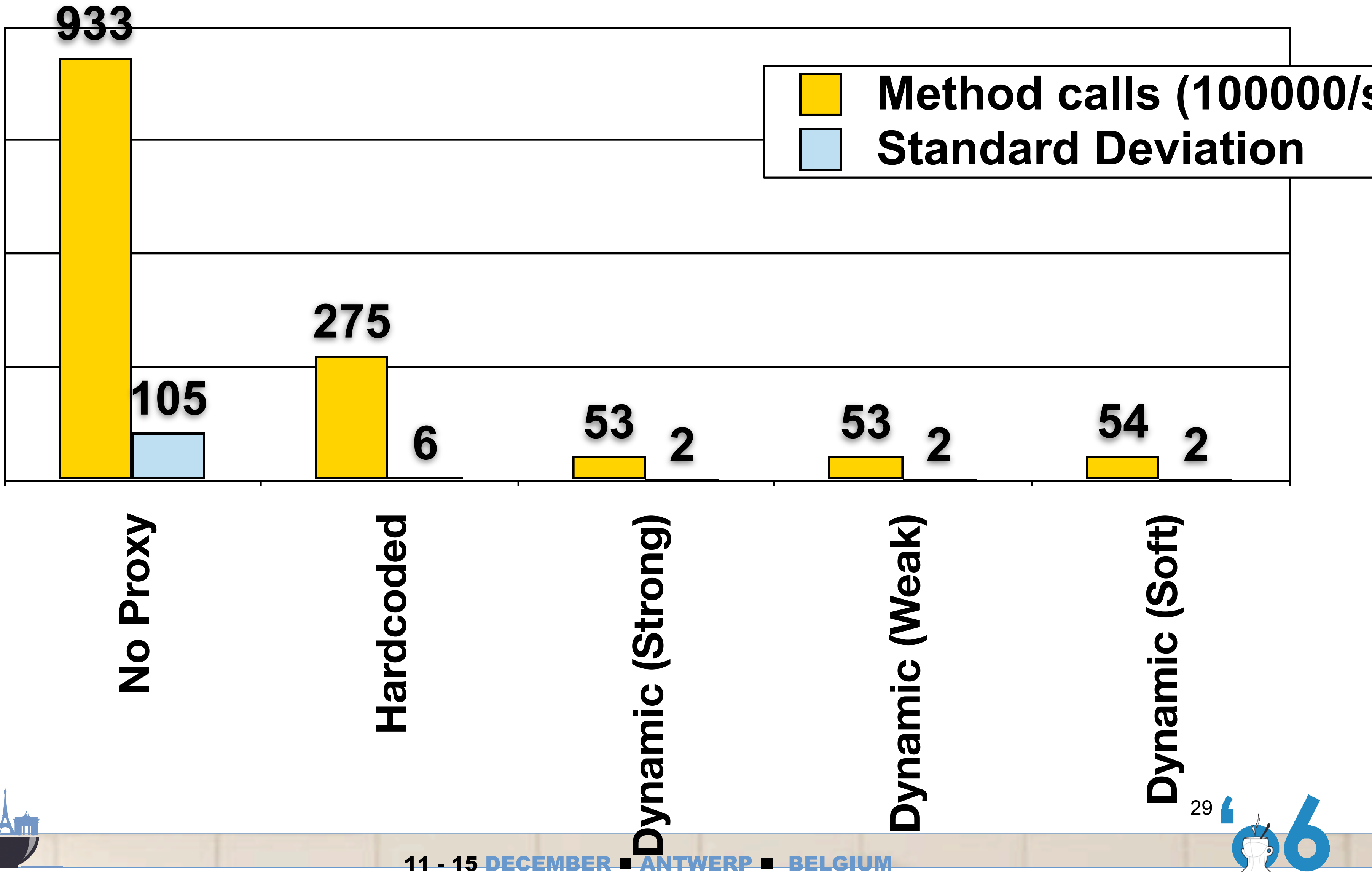
Proxy Factory

```
 public static <T> T virtualProxy(
   Class<T> subjIntf, Class<? extends T> realSubjClass) {
  return virtualProxy(subjIntf, realSubjClass, null, null);
 }


 public static <T> T virtualProxy(Class<T> subjIntf) {
  try {
   Class<?> c = Class.forName(subjIntf.getName() + "Impl");
   Class<? extends T> realSubjClass =
    c.asSubclass(subjIntf);
   return virtualProxy(subjIntf, realSubjClass);
  } catch (ClassNotFoundException e) {
   throw new IllegalArgumentException(e);
  }
 }
}
```

```java
import static java.util.concurrent.TimeUnit.SECONDS;
import static basicproxy.ProxyFactory.virtualProxy;

public class WorldMarket1 {
  public static void main(String[] args)
      throws Exception {
    Company maxsol = new Company("Maximum Solutions",
        1000 * 1000, virtualProxy(MoralFibre.class));
    SECONDS.sleep(2); // better than Thread.sleep();
    maxsol.makeMoney();
    System.out.println(maxsol);
    SECONDS.sleep(2);
    maxsol.damageEnvironment();
    System.out.println(maxsol);
    SECONDS.sleep(2);
    maxsol.becomeFocusOfMediaAttention();
    System.out.println(maxsol);
  }
}
```

```
Oh goodie!
Maximum Solutions has $ 2000000.00
Oops, sorry about that oilspill...
Maximum Solutions has $ 8000000.00
Look how good we are...
Moral Fibre Created!
Maximum Solutions has $ 7100000.00
```

Performance of Dynamic Proxies

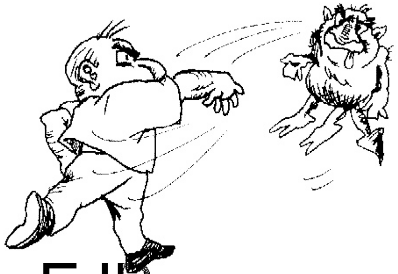


Analysis of Performance Results

- Consider performance in real-life context
  - ➲ How often is a method called per second?
  - ➲ What contention are you trying to solve – CPU, IO or memory?
    - Probably the wrong solution for CPU bound contention
- Big deviation for “No Proxy” – probably due to HotSpot compiler inlining method call

Virtual Proxy Gotchas

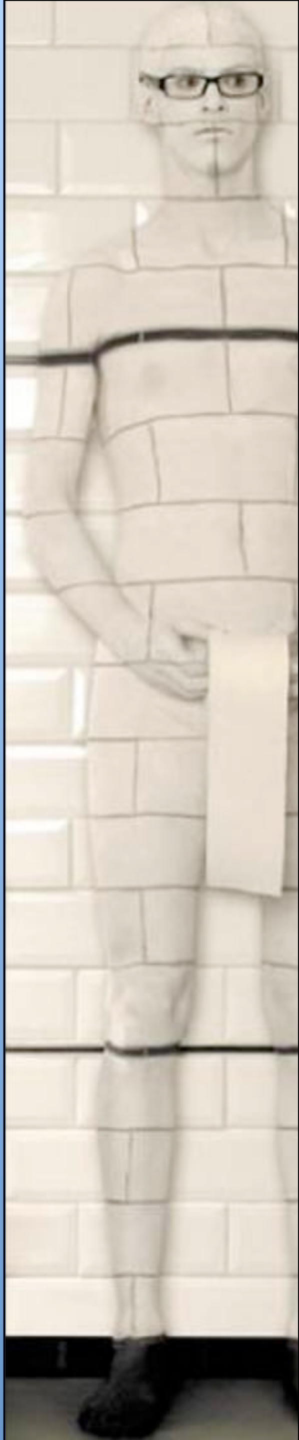
- Be careful how you implement equals()
  - Should always be symmetric (from JavaDocs):
    - For any non-null reference values x and y, x.equals(y) should return true if and only if y.equals(x) returns true
- Exceptions
  - General problem with proxies
    - Local interfaces vs. remote interfaces in EJB
  - Were checked exceptions invented on April 1st ?



Checkpoint

- We’ve looked at the concept of a *Virtual Proxy* based on the GoF pattern
- We have seen how to implement this with dynamic proxies (since JDK 1.3)
- Lastly, we were unsurprised that dynamic proxy performs worse than handcoded proxy
- Next we will look at Soft and Weak References

DEMO



References (Strong, Soft, Weak)

- We want to release references when possible
  - Saves on memory
  - Soft, Weak and Strong references offer different benefits
  - Works in conjunction with our dynamic proxy
  - However, references are not transparent

Strong, Soft and Weak References

- Java 1.2 introduced concept of soft and weak references
- Strong reference is never released
- Weak reference is released when no strong reference is pointing to the object
- Soft reference can be released, but will typically only be released when memory is low
  - Works correctly since JDK 1.4

Object Adapter Pattern – Pointers

- References are not transparent
- We make them more transparent by defining a Pointer interface
  - Can then be Strong, Weak or Soft

```
public interface Pointer<T> {
  void set(T t);
  T get();
}
```

Strong Pointer

- Simply contains a strong reference to object
  - Will never be garbage collected

```
public class StrongPointer<T>
    implements Pointer<T> {
  private T t;
  public void set(T t) { this.t = t; }
  public T get()       { return t; }
}
```

Reference Pointer

- Abstract superclass to either soft or weak reference pointer

```
import java.lang.ref.Reference;
public abstract class RefPointer<T>
    implements Pointer<T> {
  private Reference<T> ref;
  protected void set(Reference<T> ref) {
    this.ref = ref;
  }
  public T get() {
    return ref == null ? null : ref.get();
  }
}
```

Soft and Weak Reference Pointers

- Contains either soft or weak reference to object
- Will be garbage collected later

```
public class SoftPointer<T> extends RefPointer<T> {
  public void set(T t) {
    set(new SoftReference<T>(t));
  }
}

public class WeakPointer<T> extends RefPointer<T> {
  public void set(T t) {
    set(new WeakReference<T>(t));
  }
}
```

Using Turbocharged enums

- We want to define enum for these pointers
- But, we don’t want to use switch
  - Switch and multi-conditional if-else are anti-OO
  - Rather use inheritance, strategy or state patterns
- Enums allow us to define abstract methods
  - We implement these in the enum values themselves

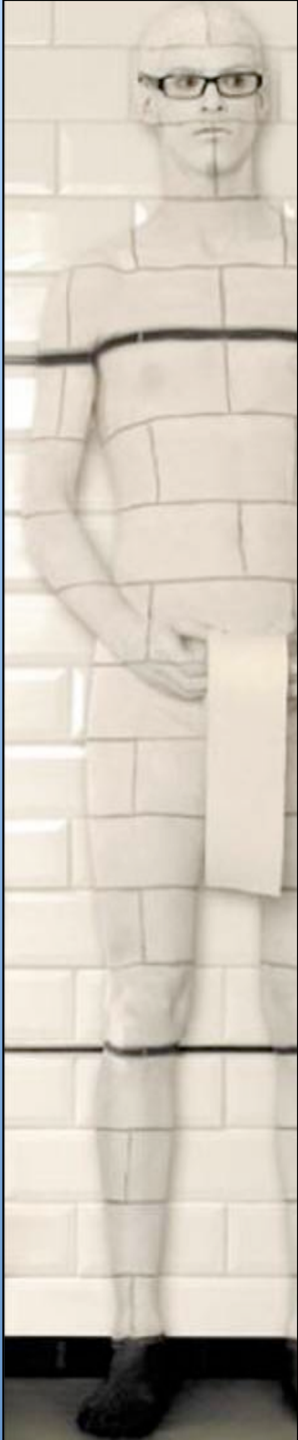
```
public enum PointerType {
  STRONG { // these are anonymous inner classes
    public <T> Pointer<T> make() { // note generics
      return new StrongPointer<T>();
    }
  },
  WEAK {
    public <T> Pointer<T> make() {
      return new WeakPointer<T>();
    }
  },
  SOFT {
    public <T> Pointer<T> make() {
      return new SoftPointer<T>();
    }
  };


  public abstract <T> Pointer<T> make();
}
```

PointerTest Example

```
public void test(PointerType type) {
  System.out.println("Testing " + type + "Pointer");
  String obj = new String(type.toString());
  Pointer<String> pointer = type.make();
  pointer.set(obj);
  System.out.println(pointer.get());
  obj = null;
  forceGC();
  System.out.println(pointer.get());
  forceOOME();
  System.out.println(pointer.get());
  System.out.println();
}
```

```
Testing STRONG Pointer
STRONG
STRONG
STRONG

Testing WEAK Pointer
WEAK
null
null

Testing SOFT Pointer
SOFT
SOFT
null
```

DEMO



Danger – References

- References put additional strain on GC
- Only use with large objects
- Memory space preserving measure
  - But can impact on performance
  - Additional step in GC that runs in separate thread

Combining Pointers and Proxies

- With dynamic proxies, we can create objects on demand
  - How can we use our Pointers to clear them again?

```
import java.lang.reflect.*;

public class VirtualProxy implements InvocationHandler {
  private final Pointer<Object> realSubjectPointer;
  private final Object[] constrParams;
  private final Constructor<?> subjectConstr;

  public VirtualProxy(Class<?> realSubjectClass,
                      Class[] constrParamTypes,
                      Object[] constrParams,
                      PointerType pointerType) {
    try {
      subjectConstr = realSubjectClass.
        getConstructor(constrParamTypes);
      realSubjectPointer = pointerType.make();
    } catch (NoSuchMethodException e) {
      throw new IllegalArgumentException(e);
    }
    this.constrParams = constrParams;
  }
  private Object realSubject() throws Throwable {
    synchronized (this) {
      Object realSubject = realSubjectPointer.get();
      if (realSubject == null) {
        realSubject = subjectConstr.newInstance(
          constrParams);
        realSubjectPointer.set(realSubject);
      }
      return realSubject;
    }
  }
  public Object invoke(Object proxy, Method method,
                Object[] args) throws Throwable {
    return method.invoke(realSubject(), args);
  }
}
```

- We now use the PointerType to create either strong, soft or weak references

Weak Pointer is cleared when we don’t have a strong ref

```
Company maxsol = new Company("Maximum Solutions",
    1000000, virtualProxy(MoralFibre.class, WEAK));
SECONDS.sleep(2);
maxsol.damageEnvironment();
maxsol.becomeFocusOfMediaAttention();
```

```
Oops, sorry about that oilspill...
Look how good we are...
Moral Fibre Created!
Oops, sorry about that oilspill...
Look how good we are...
Moral Fibre Created!
```

```
// short term memory...
System.gc();
SECONDS.sleep(2);
maxsol.damageEnvironment();
maxsol.becomeFocusOfMediaAttention();
```

Soft Pointer more appropriate

```java
Company maxsol = new Company("Maximum Solutions",
    1000000, virtualProxy(MoralFibre.class, SOFT));
SECONDS.sleep(2);
maxsol.damageEnvironment();
maxsol.becomeFocusOfMediaAttention();

System.gc(); // ignores soft pointer
SECONDS.sleep(2);
maxsol.damageEnvironment();
maxsol.becomeFocusOfMediaAttention();

forceOOME(); // clears soft pointer
SECONDS.sleep(2);
maxsol.damageEnvironment();
maxsol.becomeFocusOfMediaAttention();
}
private static void forceOOME() {
  try {byte[] b = new byte[1000 * 1000 * 1000];}
  catch (OutOfMemoryError error)
  { System.err.println(error); }
}
```

```
Oops, sorry about that oilspill...
Look how good we are...
Moral Fibre Created!
Oops, sorry about that oilspill...
Look how good we are...
java.lang.OutOfMemoryError:
    Java heap space
Oops, sorry about that oilspill...
Look how good we are...
Moral Fibre Created!
```

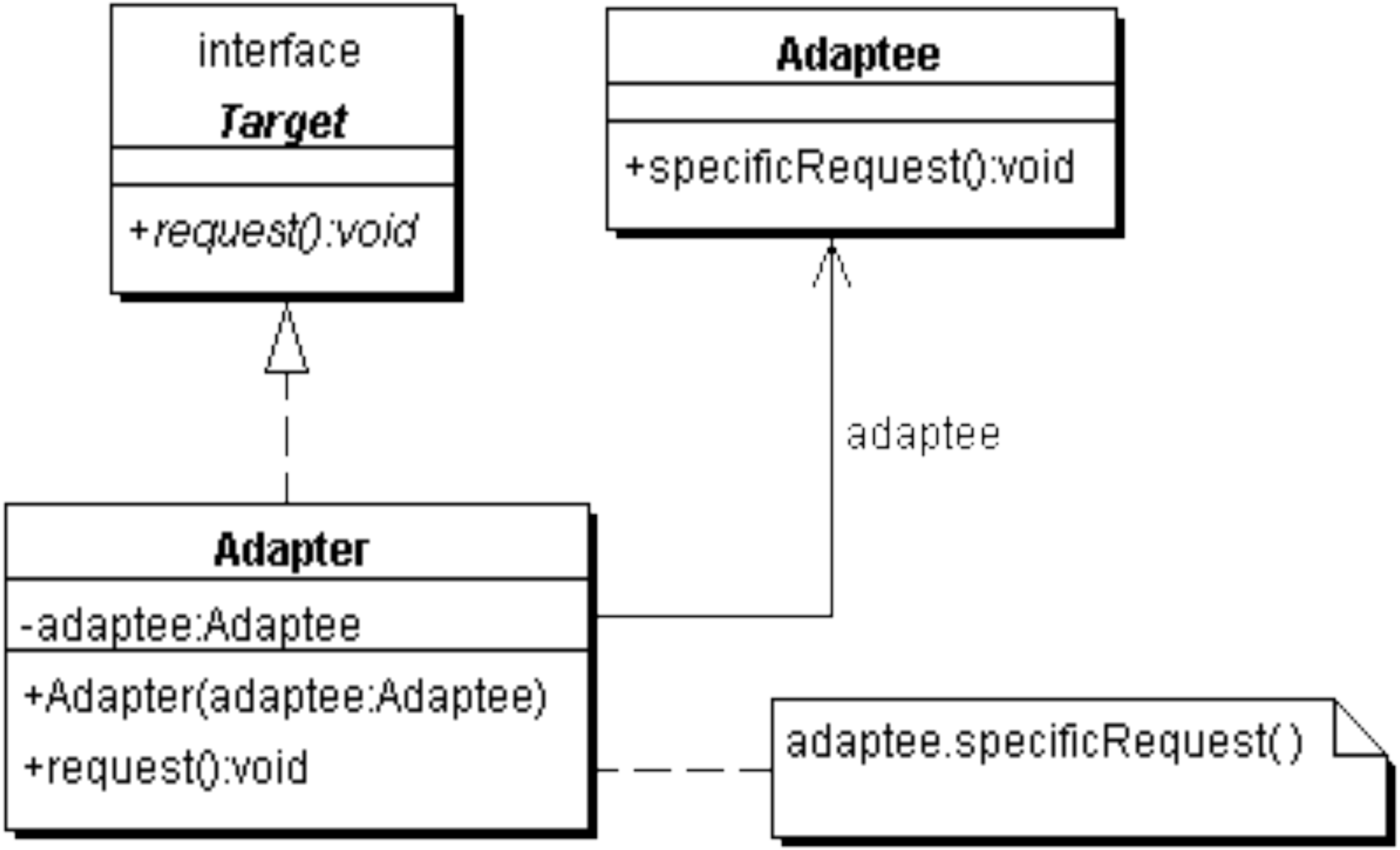
Further uses of Dynamic Proxy

- Protection Proxy
  - Only route call when caller has correct security context
    - Similar to the “Personal Assistant” pattern
- Dynamic Decorator or Filter
  - We can add functions dynamically to an object
  - See newsletter # 34
  - Disclaimer: a bit difficult to understand

Dynamic Object Adapter

- Based on Adapter pattern by GoF
- Plain Object Adapter has some drawbacks:
  - Sometimes you want to adapt an interface, but only want to override some methods
  - E.g. java.sql.Connection
- Structurally, the patterns Adapter, Proxy, Decorator and Composite are almost identical

Object Adapter Structure (GoF)



- We delegate the call if the adapter has a method with this signature
- Objects adaptee and adapter can be of any type

```
public Object invoke(Object proxy, Method method,
                     Object[] args) throws Throwable {
  try {
    // find out if the adapter has this method
    Method other = adaptedMethods.get(
      new MethodIdentifier(method));
    if (other != null) { // yes it has
      return other.invoke(adapter, args);
    } else { // no it does not
      return method.invoke(adaptee, args);
    }
  } catch (InvocationTargetException e) {
    throw e.getTargetException();
  }
}
```

- The ProxyFactory now gets a new method:

```
public class ProxyFactory {
   public static <T> T adapt(Object adaptee,
                             Class<T> target,
                             Object adapter) {
    return target.cast(Proxy.newProxyInstance(
  Thread.currentThread().getContextClassLoader(),
        new Class[] {target},
        new DynamicObjectAdapter(
          adapter, adaptee)));
  }
}
```

- Client can now adapt interfaces very easily

```
import static com.maxoft.proxy.ProxyFactory.*;

// ...

Connection con = DriverManager.getConnection("...");
Connection con2 = adapt(con, Connection.class,
  new Object() {
    public void close() {
     System.out.println("No, don’t close connection");
    }
  });
```

- For additional examples of this technique, see The Java Specialists’ Newsletter # 108
  - ➲ http://www.javaspecialists.eu

Benefits of Dynamic Proxies

- Write once, use everywhere
- Single point of change
- Elegant coding on the client
  - Esp. combined with static imports & generics
- Slight performance overhead
  - But view that in context of application

Conclusion

- Dynamic proxies can make coding more consistent
  - Reduce WET
    - Write Every Time
- Easy to use, once syntax is understood
- Παν Μετρον Αριστον
  - Everything in moderation!

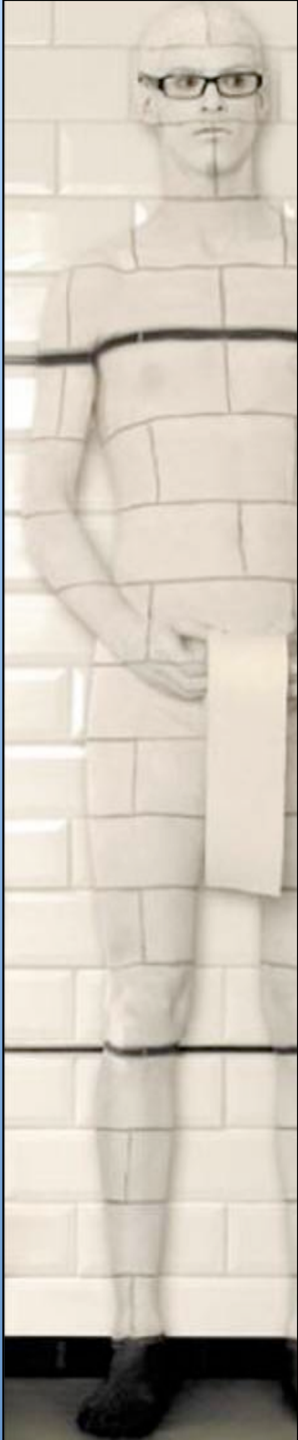
“How can I become a Java Specialist?”

1. Read the JVM Specification
2. Read the Java Language Specification
3. Read Brian Goetz book on concurrency
4. Read source code of libraries you use

'06 JAVAPOLIS

11 - 15 DECEMBER ■ ANTWERP ■ BELGIUM

Q&A



'06 JAVAPOLIS

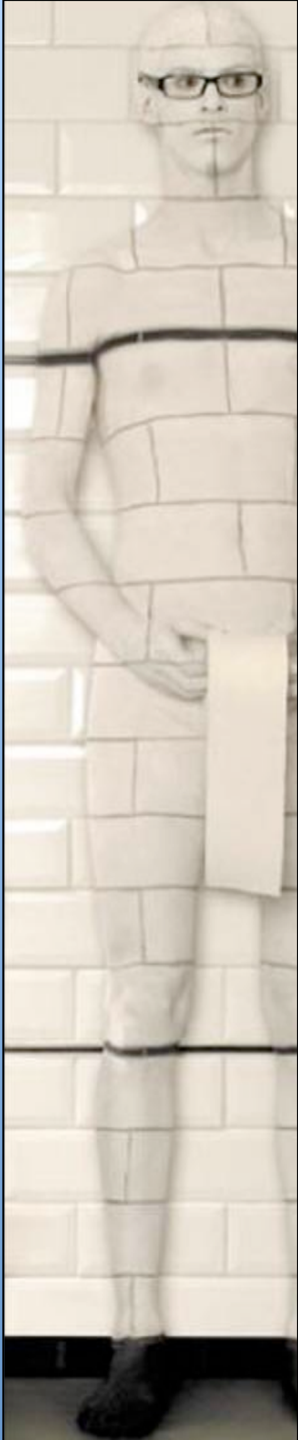
11 - 15 DECEMBER ■ ANTWERP ■ BELGIUM

Java Specialists in Action

Dr Heinz Kabutz

The Java Specialists' Newsletter

heinz@javaspecialists.eu



Thank you for your attention!

Adobe®

JBoss® a division of Red Hat

ORACLE®

Sun® microsystems

The Network is the Computer™