

# Live Productive Coder

**Dr Heinz M. Kabutz**

Last modified 2015-05-15

**[www.javaspecialists.eu](http://www.javaspecialists.eu)**



**Javaspecialists.eu**  
java training

© 2007-2015 Heinz Max Kabutz – All Rights Reserved

# Background

- **Dr Heinz Kabutz**

- Lives in Χωραφάκια, Χανιά
- The Java Specialists' Newsletter
  - 70 000 readers in 134 countries
  - <http://www.javaspecialists.eu>
- Java Champion



**Java**<sup>™</sup>  
Champions



# Productive Coder

**How you can have more fun interacting  
with your machine ...**

**... and make your computer less  
frustrated with having you as operator**

# Human vs Computer



# Machine.join()

- **Typical coder works 60 hours per week**
  - Unless you're a startup, then 120 more likely
  - We all want maximum of 40 hours
- **Coder & machine should be one**
  - Feel the machine
  - Understand the machine
  - Speak nicely to the machine :-)

# Human Mind Reading

- **Human Computer Interaction is progressing slowly**
  - You should be able to type this whilst at the same time watching TV.
  - When you make a typing error, you should know that you have made it without looking at the screen

# Keyboard Skills

- **Not all coders can touch type**
  - Each keyboard has dimple for index fingers
  - Finger controls the buttons above and below it
- **Initial investment of about 20 hours**



# Avoid Point & Click Coding

- **Try to mainly use the keyboard – minimise mouse use**
  - Menu driven copy & paste ...
- **European keyboard layouts bad for coding**
  - Semicolon and curly braces
  - Use US keyboard layout and type blindly

# Keyboard Magic

- **Back to the basics of working with computers**
  - Applies to any language, not just Java
- **But, Java's IDEs make this approach even more productive**

# Keyboard Shortcuts

- **Memorise as many as possible**
  - Use them frequently
- **Every IDE is different**
  - Sometimes on purpose it seems
  - CTRL+D in IntelliJ & Eclipse
- **Learn vim**
  - Productive for small jobs
  - Good discipline in keyboard use

# Keyboard Stickers

javaspecialists.eu

<b>A</b> Find Action	<b>B</b> Goto Decl Goto Impl	<b>C</b> Extr Const View Changes	<b>D</b> Duplicate	<b>E</b> View Recent	<b>F</b> Extr Field Find	<b>G</b> Goto Line	<b>H</b> Type Hier Call Hier	<b>I</b> Impl Method Indent	<b>J</b> Template... Surround...	<b>K</b> VCS Commit	<b>L</b> Reformat	<b>M</b> Extr Meth Scroll to Center
<b>N</b> Goto class Goto file	<b>O</b> Override Meth Optim Imports	<b>P</b> Extr Param Param Info	<b>Q</b> Quick Docs Context Info	<b>R</b> Replace Repl Struct	<b>S</b> Search Struct Project Struct	<b>T</b> Surround VCS Upd	<b>U</b> Goto Super Togg Case	<b>V</b> Intr Var Paste Recent	<b>W</b> Sel Succ Desel Succ	<b>X</b> Cut	<b>Y</b> Delete Line Sync	<b>Z</b> Undo Redo
<b>0</b> Msg Win	<b>1</b> Proj Win	<b>2</b> Comm Win	<b>3</b> Find Win	<b>4</b> Run Win	<b>5</b> Debug Win	<b>6</b> Todo Win	<b>7</b> Struct Win	<b>8</b> Hier Win	<b>9</b> Changes Win	Insert Generate...	Home Nav Bar	Delete Del to End Safe Delete
Esc Goto Editor Hide Active	<b>F1</b> Quick Doc Err Desc	<b>F2</b> Next Error Prev Err	<b>F3</b> Find Next Find Prev	<b>F4</b> Goto Source Close Editor	<b>F5</b> Copy Clone	<b>F6</b> Move Rename	<b>F7</b> Step Into Find usages	<b>F8</b> Step Over Step Out	<b>F9</b> Resume Compile	<b>F10</b> Run Sel & Run	<b>F11</b> Bookmark Fullscreen	<b>F12</b> Prev Win File Struct
← Last Edit Loc Del to start	/ Comment	PgUp Top	PgDn Bottom	[ Sel to Start Block Start	] Sel to End Block End	+ Expand Expand All	- Collapse Collapse All	← Prev Tab Prev Word	→ Next Tab Next Word	↑ Prev Meth Line Up	↓ Next Meth Line Down	· VCS Popup Switch Scheme
Enter Intentions Statement Complete	Shift				Ctrl +Alt	Tab Switcher	Alt +Ctrl					
Backspace Code Complete Smart Complete Class Complete												

# Dragon Naturally Speaking

- **“Type” at 100 words per minute**
- **Useful for JavaDocs**
- **Challenging in noisy cubevile office environment**
  - or home office “papa, can I play with the ipad?”

# The Right Kind of Lazy



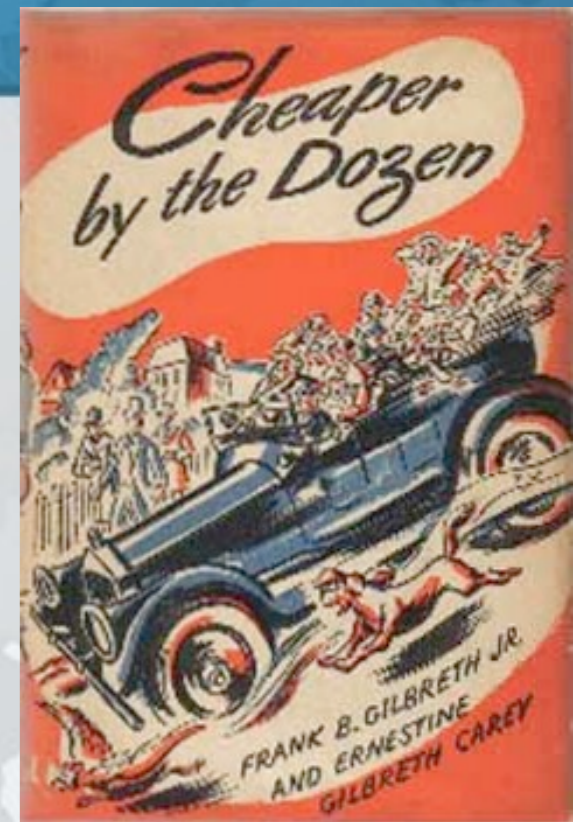
# “Cheaper by the Dozen”

- **Book from 1948**

- Story of efficiency experts
- Always studied the laziest factory worker

- **In coding we want good kind of lazy**

- Too lazy - Benny Darren\*
- Not lazy enough - George Happy\*
- Think Lazy - Sascha Schafskopf\*



\* names changed to protect the guilty

# Benny Darren - Too Lazy

- **The Copy & Paste Programmer**
  - Extremely “productive”
  - Crazy lines of code per day (LOC)
  - Features produced at alarming speed
- **And then the bug reports came in ...**



# George Happy - !Lazy Enough

- **Worked in a non-IT job before**
  - But took “typing” as a school subject
- **Code burned my eyes**
  - Deleted!!!

# Sascha Schafskopf

- **Coded without thinking**
  - Resulting code was overly complex
  - Didn't bother learning the Java API

```
public void removeAlarmContainerFromTable(
    AlarmContainer ac) {
    int i;
    HistoricalAlarmContainer h=null;
    for (i=0; i<rows.size(); i++) {
        h= getAlarmContainer(i);
        if (h.getAlarmInfo().getUniqueID()
            ==ac.getAlarmInfo().getUniqueID())
            i=Integer.MAX_VALUE-1;
    }
    if (i==Integer.MAX_VALUE) {
        rows.removeElement(h);
    }
}
```

# Sorting by toString() Value

```
public Vector sortVector (Vector unsorted) {
    Vector sorted = new Vector();
    Vector sortingVector = new Vector();
    for (int i=0; unsorted.size() > i ; i++) {
        String temp = unsorted.get(i).toString();
        sortingVector.add(temp);
    }
    Collections.sort(sortingVector);
    sortingVector.trimToSize();
    for (int i=0; i < sortingVector.size(); i++) {
        for (int j=0; j < sortingVector.size(); j++) {
            if (sortingVector.get(i) == unsorted.get(j).toString()) {
                sorted.add(i, unsorted.get(j));
            }
        }
    }
    return sorted;
}
```

# Horrible, but at least correct

```
public <E> Vector<E> sortVector(Vector<E> unsorted) {  
    Vector<E> sorted = new Vector<>(unsorted);  
    Collections.sort(sorted, new Comparator<E>() {  
        public int compare(E e1, E e2) {  
            return String.valueOf(e1).compareTo(String.valueOf(e2));  
        }  
    });  
    return sorted;  
}
```

# Java 8 Lambdas - Yippee!

- **Better or worse? Help me decide!**

```
public <E> Vector<E> sortVector(Vector<E> unsorted) {  
    Vector<E> sorted = new Vector<>(unsorted);  
    Collections.sort(sorted, (e1, e2) ->  
        String.valueOf(e1).compareTo(String.valueOf(e2)));  
    return sorted;  
}
```

# Java 8 Comparator.comparing()

- **Knowing API leads to shorter, cleaner code**

```
public <E> Vector<E> sortVector(Vector<E> unsorted) {  
    Vector<E> sorted = new Vector<>(unsorted);  
    Collections.sort(sorted,  
        Comparator.comparing(String::valueOf));  
    return sorted;  
}
```

# Know Your IDE

- **No matter if IDEA, Eclipse or Netbeans**
- **Short coding demo**
  - Extreme Java - Concurrency Course Exercise
    - synchronized to ReentrantReadWriteLock
  - First attempt is by hand, but using IDE
  - Next attempt is easier - live code templates
  - Next with lambdas and then try-with-resource

# Quick Demo





# Coding with Lambda Idioms



# Lambda Locking Idioms

```
public static <T> T lock(Lock lock, Supplier<T> task) {  
    lock.lock();  
    try {  
        return task.get();  
    } finally {  
        lock.unlock();  
    }  
}
```

```
public static void lock(Lock lock, Runnable task) {  
    lock.lock();  
    try {  
        task.run();  
    } finally {  
        lock.unlock();  
    }  
}
```

# ReadWriteLock Idioms

```
public static <T> T readLock(  
    ReadWriteLock rwlock, Supplier<T> task) {  
    return lock(rwlock.readLock(), task);  
}
```

```
public static void readLock(  
    ReadWriteLock rwlock, Runnable task) {  
    lock(rwlock.readLock(), task);  
}
```

```
public static <T> T writeLock(  
    ReadWriteLock rwlock, Supplier<T> task) {  
    return lock(rwlock.writeLock(), task);  
}
```

```
public static void writeLock(  
    ReadWriteLock rwlock, Runnable task) {  
    lock(rwlock.writeLock(), task);  
}
```

# LambdaReadWriteLock

```
public class LambdaReadWriteLock {
    private final ReentrantReadWriteLock rwlock;

    public LambdaReadWriteLock(
        ReentrantReadWriteLock rwlock) {
        this.rwlock = rwlock;
    }

    public <T> T readLock(Supplier<T> task) {
        return LockIdioms.readLock(rwlock, task);
    }

    // etc.
}
```

# writeLock() Deadlock Check

```
public void writeLock(Runnable task) {
    checkThatWeDoNotHoldReadLocks();
    LockIdioms.writeLock(rwlock, task);
}

private void checkThatWeDoNotHoldReadLocks() {
    if (rwlock.getReadHoldCount() != 0) {
        throw new IllegalMonitorStateException(
            "trying to upgrade read to write");
    }
}
```

# Lambda Idiom Meet Exception

- Lambdas do not “play nice” with checked exceptions

```
LockIdioms.lock(lock, () -> Thread.sleep(10));
```

- Callable can easily result in ambiguity

# Java 7 “Try-With-Resource”

```
public class LockResource implements AutoCloseable {
    private final Lock lock;

    public LockResource(Lock lock) {
        this.lock = lock;
    }

    public LockResource lock() {
        lock.lock();
        return this;
    }

    public void close() {
        lock.unlock();
    }
}
```

# Try-With-Resource

- **Program flow is no longer interrupted by lambda context**

```
try (LockResource lr = lock.lock()) {  
    Thread.sleep(10);  
}
```



# More Lambda Idioms

- **StampedLock introduced in Java 8**
  - Described in [JavaSpecialists.eu](http://javaspecialists.eu) - issue 215
- **Allows**
  - pessimistic exclusive locks (write)
  - pessimistic non-exclusive locks (read)
  - optimistic read with good collision detection
- **Idioms are much harder than Lock**

# Design Patterns



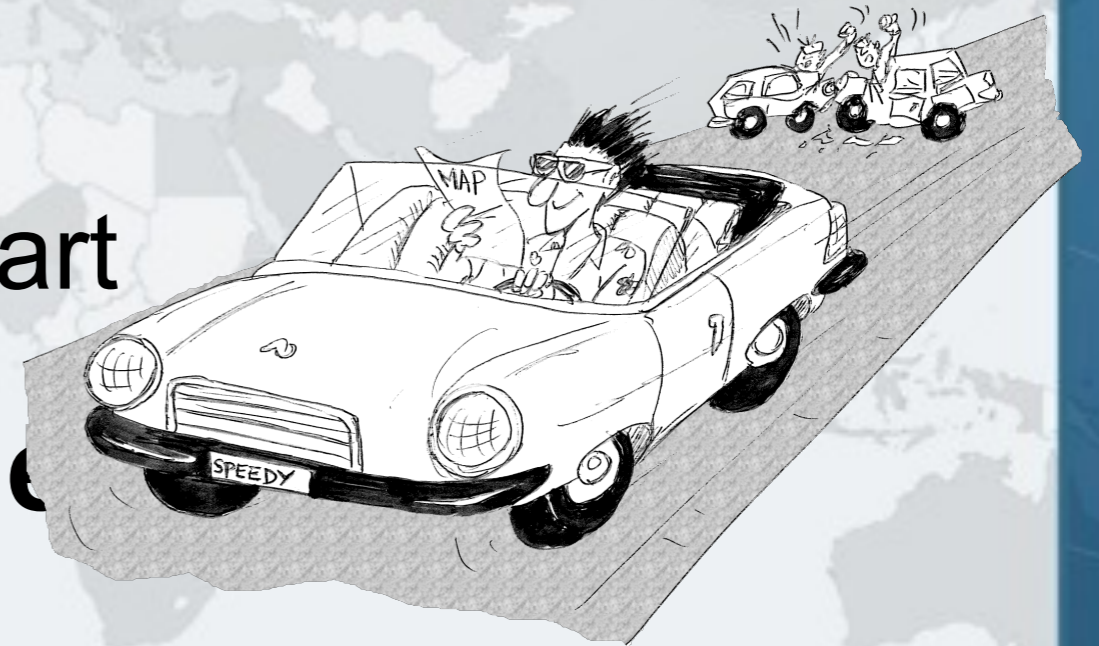
# Fingers Overtaking the Brain

- **You still need to plan**

- Stop & think before you start

- **When shortcuts & fingers are too fast:**

- Increase speed of your brain
- Think in higher level concepts, such as Design Patterns

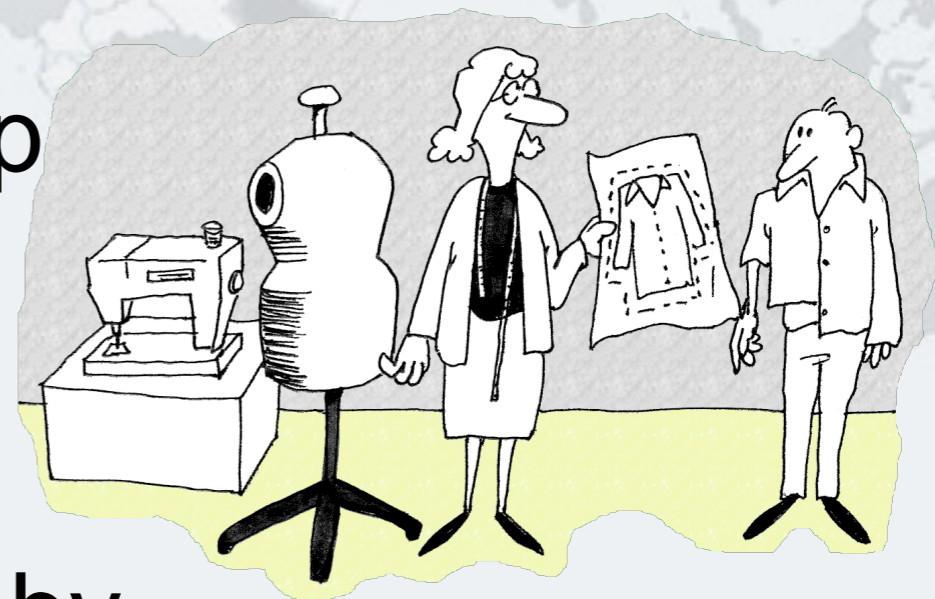


# Design Patterns

- **Mainstream of OO landscape, offering**

**us:**

- View into brains of OO exp
- Quicker understanding of existing designs
  - e.g. Visitor pattern used by Annotation Processing Tool
- Improved communication between developers



# Vintage Wines



- **Design Patterns are like good red**
  - You cannot appreciate them at first
  - As you study them you learn the difference between plonk and vintage, or bad and good designs
  - As you become a connoisseur you experience the various textures you didn't notice before
- **Warning: Once you are hooked, you will no longer be satisfied with inferior**

# Refactoring

How to shoot yourself in your foot in style



# “Houston, We Have a Problem”

- **“Our lead developer has left”**
  - Software works most of the time
  - We have to fix it, and add some features ...



# How do you start?

- **Ask some basic questions**

- What code is dead?
  - Stories of whole teams working on dead code for years
- Where are the unit test?
- Where could access control be tighter?
- What portion of code is commented?
- How can I find bad code? Copy & paste code?



# Initial Investigation

- **Check where comments are missing**
  - Doclet in Newsletter 049
- **Find fields that are not private**
  - Doclet in Newsletter 035

# Initial Investigation

- **Count # of classes, lines of code each**
  - Aim for average of less than 100 lines per class
  - One of my customers had one Java class > 30000 LOC
- **Code coverage tool against unit tests**
  - JaCoCo by Marc Hoffmann

# What are Realistic Values?

	<b># Classes</b>	<b>Total LOC AVG/STDEV</b>	<b>Uncommented Elements</b>
<b>Project 1 South Africa</b>	<b>1359</b>	<b>263790 194 / 337</b>	<b>24291 18 per class</b>
<b>Project 2 Germany</b>	<b>442</b>	<b>62393 141 / 149</b>	<b>7298 17 per class</b>
<b>Ideal</b>	<b>1000</b>	<b>80260 80 / 61</b>	<b>1000 max 1 per class</b>

- **Beware, LOC is only a rough guess**

# Comments must Explain “Why”

- **Comment tips**

- Should not just be: Method getName returns the name.
- Turn off automatic comment generation
- Either proper comments, or leave them out
- Method names and parameters should be descriptive

# Comments must Explain “Why”

- **“Why I don’t read your code comments ...”**
  - Most misunderstood newsletter - Issue 039
  - I do write my own comments, but about “why” not “what”
  - seldom find projects with well-written comments

# Comments: j.a.c.ColorSpace

- **Rather insightful comment in JDK 1.3:**

```
/**
 * Returns the name of the component given the
 * component index
 */
public String getName(int idx) {
    /* REMIND – handle common cases here */
    return new String(
        "Unnamed color component(" + idx + ")");
}
```

- **What is “REMIND” supposed to tell us?**

# Comments: j.a.c.ColorSpace

- **JDK 1.4: more text, still the question**

```
/**
 * Returns the name of the component given the
 * component index.
 *
 * @param idx The component index.
 * @return The name of the component at the
 * specified index.
 */
public String getName(int idx) {
    /* REMIND - handle common cases here */
    return new String(
        "Unnamed color component(" + idx + ")");
}
```

# Java 5

```
/** Returns the name of the component given the  
 * component index.  
 * @param idx The component index.  
 * @return The name of the component at the  
 * specified index.  
 * @throws IllegalArgumentException if idx is less  
 * than 0 or greater than numComponents - 1 */  
public String getName (int idx) {  
    /* REMIND - handle common cases here */  
    if ((idx < 0) || (idx > numComponents - 1)) {  
        throw new IllegalArgumentException(  
            "Component index out of range: " + idx);  
    }  
    return new String(  
        "Unnamed color component("+idx+")");  
}
```



# Java 6 onwards

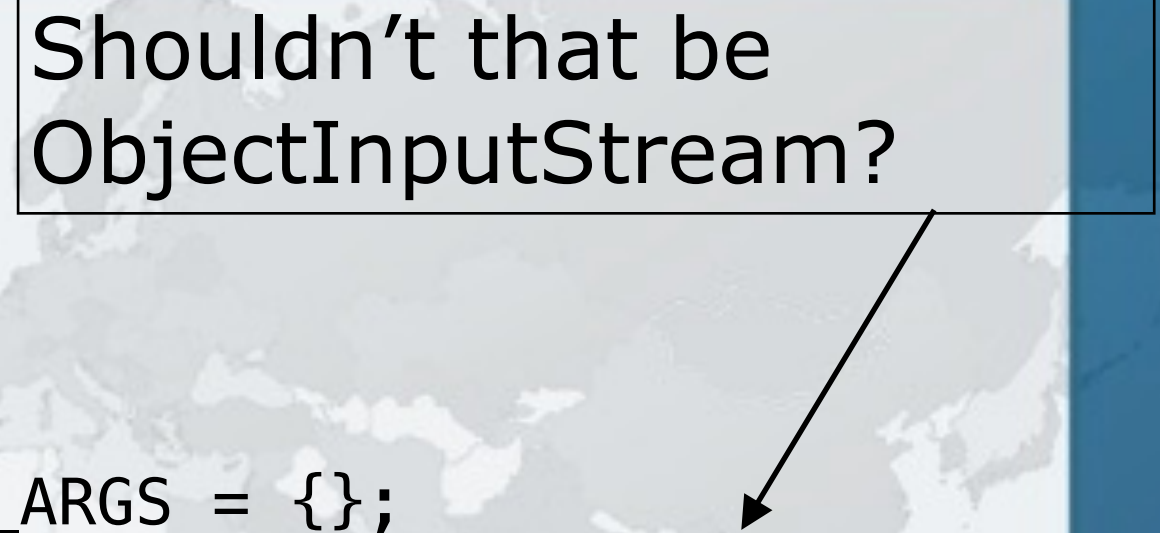
```
/** Returns the name of the component given the  
 * component index.  
 * @param idx The component index.  
 * @return The name of the component at the  
 * specified index.  
 * @throws IllegalArgumentException if idx is less  
 * than 0 or greater than numComponents - 1 */  
public String getName (int idx) {  
    /* REMIND - handle common cases here */  
    if ((idx < 0) || (idx > numComponents - 1)) {  
        throw new IllegalArgumentException(  
            "Component index out of range: " + idx);  
    }  
    if (compName == null) {  
        switch (type) {  
            case ColorSpace.TYPE_XYZ:  
                compName = new String[] {"X", "Y", "Z"}; break;
```

# Commenting Out Code

- **Source Control Systems**
  - Have been around for decades
- **Don't duplicate source control work**
- **If code is dead, delete it, don't comment it out**

# Funny Comments

Shouldn't that be  
ObjectInputStream?



- **JDK 1.3: java.io.ObjectStreamClass**

```
private final static Class[] NULL_ARGS = {};  
//WORKAROUND compiler bug with following code.  
//static final Class[] OIS_ARGS={ObjectInputStream.class};  
//static final Class[] OOS_ARGS={ObjectOutputStream.class};  
private static Class[] OIS_ARGS = null;  
private static Class[] OOS_ARGS = null;  
private static void initStaticMethodArgs() {  
    OOS_ARGS = new Class[1];  
    OOS_ARGS[0] = ObjectOutputStream.class;  
    OIS_ARGS = new Class[1];  
    OIS_ARGS[0] = ObjectInputStream.class;  
}
```

- **“The compiler team is writing useless code again ...”**

– <http://www.javaspecialists.eu/archive/Issue046.html>

# “Wonderfully Disgusting Hack”

- **JDK 1.4: java.awt.Toolkit**

```
static boolean enabledOnToolkit(long eventMask) {  
    // Wonderfully disgusting hack for Solaris 9
```

- **This made me think:**

- All software contains hacks.
- I would prefer to know about them.
- Only a real developer would write "hack" into his comments.
- Rather use Java than black-box proprietary solution with hundreds of undocumented hacks

# Before You Change Code...

- **Refactoring is dangerous!**
- **You must have good unit tests**
  - And great skill if you don't have unit tests...
- **Also system tests**
- **In troubled projects, unit tests often absent**

# Automatic Refactoring in IDEs

- **IDEs tempt us to refactor code quickly**
  - But result might be incorrect
- **Be careful, very careful**
  - Inlining is not always correct
  - Method extraction is not always correct
  - Replace duplicate code snippet is not always correct

# Automatic Tools and Reflection

- **Java tools rely on static compilation of classes**
- **Be careful when using Reflection and Dynamic Proxies**

# Check your code

- **Regularly check your own work:**

- Elements are properly commented
- Exceptions are handled correctly
- Fields are private
- Fields are final where possible
- Unit tests cover your code base
- Look for copy & paste code
  - Sometimes difficult to eliminate



# Develop with Pleasure!

- **Simplicity is beauty**

# Advanced Java Courses Crete

- **Extreme Java - Concurrency Performance**
  - 3 days, price of €2767.50
  - Week June 8-12, 2015
- **Java Specialist Master Course**
  - 4 days, price of €3075
  - June 23-26, 2015
- **[www.javaspecialists.eu](http://www.javaspecialists.eu)**
- **[heinz@javaspecialists.eu](mailto:heinz@javaspecialists.eu)**



# Live Productive Coder

**Dr Heinz M. Kabutz**

**[www.javaspecialists.eu](http://www.javaspecialists.eu)**



**Javaspecialists.eu**  
java training