Clean code and pitfalls in Java

Sebastian Oerding

22. Januar 2015

Scope of today

1. Goals of code
2. Wording
3. Be aware of pitfalls
4. Being punished for violating best practices
5. Generics, Strings and Threads
6. Nevermind the hardware?

Introduction

Goals of code

Writing really good code is a demanding task cause it aims at the following goals:

1. Having the specified / requested functionality
2. Performing well
3. Having / causing the minimum possible cost for the company considered the whole life cycle of it (the code not the company)

$\Rightarrow$ What does follow from these goals?

Developer requirements

To write good code we must

1. aim at being professionals
2. get some knowledge in areas related to programming (T-shaped)
3. know the own limits and communicate them

About the code we will see today

Having the small talk done, let's get to the code

1. Opinion and facts can not be always clearly distinguished
2. JDK 6
3. Focused on specific aspects
4. Should encourage to wonder about aspects not discussing them to end
5. Some artifical and most examples I have actually seen in production / delivered code

Wording is important!

Rain

```
/** Tests if it rains. */
@Test public void testRain() {
  IndustrialManager it = new SalesPerson();
  Assert.assertTrue(it.rains());
}
```

$\Rightarrow$ Is this a good test?

Rain - II

```
/** Tests whether it rains. */
@Test public void testRain() {
  IndustrialManager it = new SalesPerson();
  Assert.assertTrue(it.rains());
}
```

$\Rightarrow$ Better?

Rain - III

```
/** Verifies that a sales person rains. */
@Test public void salesPersonRains() {
  IndustrialManager it = new SalesPerson();
  Assert.assertTrue("Sales person unexpectedly
    does not rain!", it.rains());
}
```

$\Rightarrow$ Better, might not be perfect.

$\Rightarrow$ Developers should sharpen their awareness for requirements and how to write them down by reading IEEE 1998:830 or its successor.

Telling a story - I

Humans like stories. Hence we should take our time to tell a good story with our code:

```
...
if (mary.goodNight()) {
  john.reads();
  for (Tooth t : mary.teeth()) {
    t.brush();
  }
  switchTheLight();
}
...
```

$\Rightarrow$ Is this a good story?

Telling a story II

Small changes are sufficient to change the previous example into code which is really easy to understand:

```
...
if (mary.goesToBed()) {
  father.readsAFairytaleFor(mary);
  mary.brushesHerTeeth();
  father.turnsTheLightOff(mary.room());
}
...
```

Be aware of pitfalls!

The modulus operator

```
public boolean isOdd(int i) {
  return i % 2 == 1;
}
```

The modulus operator II

```
public boolean isOdd(int i) {
  return i % 2 == 1;
}
```

⇒ Returns **false** for all negative odd values.

⇒ Use `i % 2 != 0` instead.

Using numbers

We should define constants in a way easy to read

```
long MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
long MICROS_PER_DAY = 24 * 60 * 60 * 1000 * 1000;
public long getFactor() {
  return MICROS_PER_DAY / MILLIS_PER_DAY;
}
```

Using numbers II

We should define constants in a way easy to read - but must take care of overflows in cases like the shown one

```
long MILLIS_PER_DAY = 24 * 60 * 60 * 1000;
long MICROS_PER_DAY = 24 * 60 * 60 * 1000 * 1000L;
public long getFactor() {
  return MICROS_PER_DAY / MILLIS_PER_DAY;
}
```

$\Rightarrow$ All computations internally use int if not told otherwise.

$\Rightarrow$ Use capital 'L' (at the proper place) to avoid overflows.

But numbers are simple, are they?

We should define constants in a way easy to read

```
public final class DosEquis {
  public static void main(String... args) {
    char x = 'X';
    int i = 0;
    System.out.print(true ? x : 0);
    System.out.print(false ? i : x);
  }
}
```

$\Rightarrow$ What does the program print?

How many iterations

After the overflow and the suprising cast let us confine to a simple loop

```
public final class Count {
  public static void main(String... args) {
    final int START = 2000000000;
    int count = 0;
    for (float f = START; f < START + 50; f++)
      count++;
    System.out.println(count);
  }
}
```

$\Rightarrow$ What does the program print?

Simple comparison

The previous examples had been tricky, now let us do a simple comparison.

```
while (i != i) {}
while (i != i + 0) {}
```

$\Rightarrow$ Can we turn these loops into infinite loops?

Concatenating strings

Let us assume that we provide a convenience method to print highway names.

```
private void printHighwayName(Character hChar,
    int hNumber, String country) {
 System.out.println(hChar + hNumber + country);
}
```

Concatenating strings

Having our beautiful convenience method the following test gives us an unexpected result.

```
@Test public void printA1De() {
  printHighwayName('A', 1, "DE");
}
```

$\Rightarrow$ Results in '66DE'.

$\Rightarrow$ The CHARACTER is unboxed and the string is evaluated from left to right leading to the addition of 'A' and '1'.

Apply best practices (if appropriate)!

Punishment for violating best practices

In the boardgame Go players should not make overplays as the expected result is worse for them compared to a correct move. In Java there is also a lot of stuff we can do but we should not as the risks never pay out in the long run.

Similar to this we should expect our code to encounter situations where the fast hacks cause real problems. This ranges from problems extending or reusing the code to bugs that occur especially under heavy load in production environments and were not spotted by tests.

Confine to using common idioms

Even simple code can surprise developers if using uncommon idioms.

```
private void printWelcome() {
 byte[] message = {1, 2, 3};
 if (false)
 System.out.println(new String(message));
}
```

⇒ What could cause the message to be surprisingly printed out?

Deceiving but still surprising at a first glance

We should avoid code which sounds deceivingly simple but does not behave as expected.

```
public class Person {
  private String fName;
  private String lName;

  public boolean equals(Person p) {
    return fName.equals(p.fName) &&
      lName.equals(p.lName);
  }
}
```

$\Rightarrow$ Assuming setters / constructors to be in place this surprisingly may result in 'Max Muster' being unequal to 'Max Muster'. How?

Be slow to be fast

Never return **null** for arrays or collections.

```
public List<byte[]> renderImages(Scene scene) {
  if (renderProcess == null) return null;
  return renderProcess.render(scene);
}
```

⇒ Is no or close to no performance gain
⇒ Causes a lot of unnecessary code
⇒ Increases cyclomatic complexity of code calling this method
⇒ Requires an additonal test whereever this code is called
⇒ Makes the calling code more clumsy, more difficult to read and to perform worse

Be slow to be fast II

By a simple change all of this hassle vanishes.

```
public List<byte[]> renderImages(Scene scene) {
  return renderProcess == null ?
    Collections.<byte[]>emptyList :
    renderProcess.render(scene);
}
```

⇒ Avoids the problems

⇒ No performance loss as the list is instantiated only once

⇒ No performance loss as there is no type cast at runtime

Be slow to be fast III

It looks like a similar methods can easily be written for arrays:

```
private static final Object[] EMPTY_ARRAY = {};
public <T> T[] emptyArray() {
   //The cast is safe
   return (T[])EMPTY_ARRAY;
}
```

$\Rightarrow$ As zero sized arrays are effectively immutable the cast is safe

$\Rightarrow$ Unfortunately you will get a ClassCastException at runtime if accessing this array (for example using a for each loop)

Be slow to be fast IV

Unfortunately the approach from the previous sheet does not work in any way. However we can fix it:

```
public static <T, E extends T> T[] emptyArray(
  Class<T> theClass, Collection<E> elements) {
  return (T[]) Array.newInstance(
    theClass, elements == null ? 0 : elements.size());
}
```

$\Rightarrow$ Unfortunately this instantiates a new array for each invocation but especially on modern JVMs you should not overestimate the costs of object instantiation

Use object orientation

Even with object oriented languages it is easy and error prone to write procedural code:

```
private enum Default {A, B, C}
  public Object map(Default d) {
    switch (d) {
    case A : return null;
    case B : return null;
    case C : return null;
    ...
  }
}
```

$\Rightarrow$ It is easy to forget to extend this method if the enum is extended.

Use object orientation - II

We could fix this by having a better enum:

```
private enum Default {
  A{...}, B{...}, C{...};
  abstract Object foo();
}

public Object map(Default d) {
  return d.foo();
}
```

$\Rightarrow$ Now anybody extending the enum is enforced to implement *foo*.

Fun facts

String concatenation

Since Java 6 string concatenation is no more close to an antipattern as it has been before.

```
public void snake() {
  String s = "s";
  for (int i = 0; i < 128; i++)
    s += s;
}
```

⇒ What happens if running this code?

⇒ Probably you will have an OutOfMemoryError

⇒ If not getting the error weird stuff will happen due to duplicating the length of the string in each iteration and strings are internally **char[]** (with a length of type **int**) and the overflow of the length

Stopping Threads

Having a deeper understanding makes us to avoid some pitfalls.

```
public class C implements Runnable {
  private boolean stop;
  public void run() {
    while (!stop) {}
  }

  public void stop() {
    stop = true;
  }
}
```

$\Rightarrow$ What may happen if running this code in a Thread and concurrently resetting the flag?

Stopping Threads - II

- The loop may be (nearly) immediately left.
- The loop may be left after some time.
- The loop may never be left.

Especially the last case may sound surprising but it can happen due to a legal compiler optimization with the while loop transformed into

```
if (!stop)
  while (true) {}
```

⇒ In general the compiler may reorder instructions.
⇒ Synchronization effectively prevents some compiler optimizations concerning memory access.

Generic throws declarations

We should know about the subleteties of generics.

```
private static <T extends Throwable> T rethrow(
  Throwable t) throws T {
  throw (T)t;
}

public static void main(String[] args) {
  Throwable t = new IOException("Hi");
  PersonTest.<RuntimeException>rethrow(t);
}
```

$\Rightarrow$ What happens if trying to compile / run this code?

Should we care about the hardware / operating system?

Thoughts about hardware

Even with Java we should know some stuff about operating systems and hardware.

1. There may be different limits depending on the operating system (file handles, network sockets, ...)
2. The hardware may also reorder instructions (and most pipelined architectures will do so)
3. Writing code that causes the branch prediction of pipelined processors to fail often may drastically slow down the application / machine
4. Disabling caching by unecessary synchronization / unecessarily volatile variables may drastically slow down the application / machine

- *Clean Code Developer*
  `http://www.clean-code-developer.de/`
- Cormen, Leierson, Rivest, Stein *Introduction to algorithms* 2009; ISBN-13: 978-0262533058; MIT Press
- Joshua Bloch *Effective Java* 2008; ISBN-13: 978-0321356680; Addison-Wesley Longman
- Joshua Bloch *How to design a good API and why it matters* 2007; `http://www.youtube.com/watch?v=aAb7hSCtvGw`
- Joshua Bloch, Neil Gafter *Java Puzzlers* 2005; ISBN-13: 978-032133678; Addison-Wesley Longman
- Frank Dunkel *Projektmanagement* 2012; iX 2/2013, S. 96 ff.; Heise Verlag
- Horst Eidenberger *Softwarequalität* 2012; iX 2/2013, S. 132 ff.; Heise Verlag

- Goetz, Bloch, Bowbeer, Lea, Holmes, Peierls *Java Concurrency in Practice* 2006; ISBN-13: 978-0321349606; Addison-Wesley Longman
- Elisabeth Heinemann *Jenseits der Programmierung* 2010; ISBN-13: 978-3446422605; Carl Hanser Verlag GmbH & Co. KG
- Kevlin Henney *97 Things Every Programmer Should Know* 2010; ISBN-13: 978-0596809485; O'Reilly & Associates
- IEEE, *IEEE Recommended Practice for Software Requirements Specifications*; 1998;
- ISO/IEC/IEEE, *Systems and software engineering – Life cycle processes –Requirements engineering* E-ISBN: 978-0-7381-6591-2, IEC/ISO/IEEE
- Angelika Langer, Klaus Kreft *Java Core Programmierung* 2011; ISBN-13: 978-3-86802-075-5; entwickler.press

- Robert C. Martin *Clean Code* 2009; ISBN-13: 978-3826655487; mitp
- Oracle *How to Write Doc Comments for the Javadoc Tool* `http://www.oracle.com/technetwork/java/javase/documentation/index-137868.html`
- Oracle *The Java Tutorials* `http://docs.oracle.com/javase/tutorial`
- Andreas Spillner *Basiswissen Softwaretest* 2012; ISBN-13: 978-3864900242; dpunkt.verlag GmbH
- Kathy Sierra, Bert Bates *Sun Certified Programmer for Java 6 Study Guide* 2008; ISBN-13: 978-0071591065; Mcgraw-Hill Publ.Comp.
- Stefan Tilkov *REST und HTTP* 2011; ISBN-13: 978-3898647328; dpunkt.verlag GmbH

 Kageyama Toshiro *Lessons in the fundamentals of Go* 1998; ISBN-13: 978-4906574285; Kiseido Pubn Co