Java Coding Standards

Jörg Steffen, DFKI

steffen@dfki.de

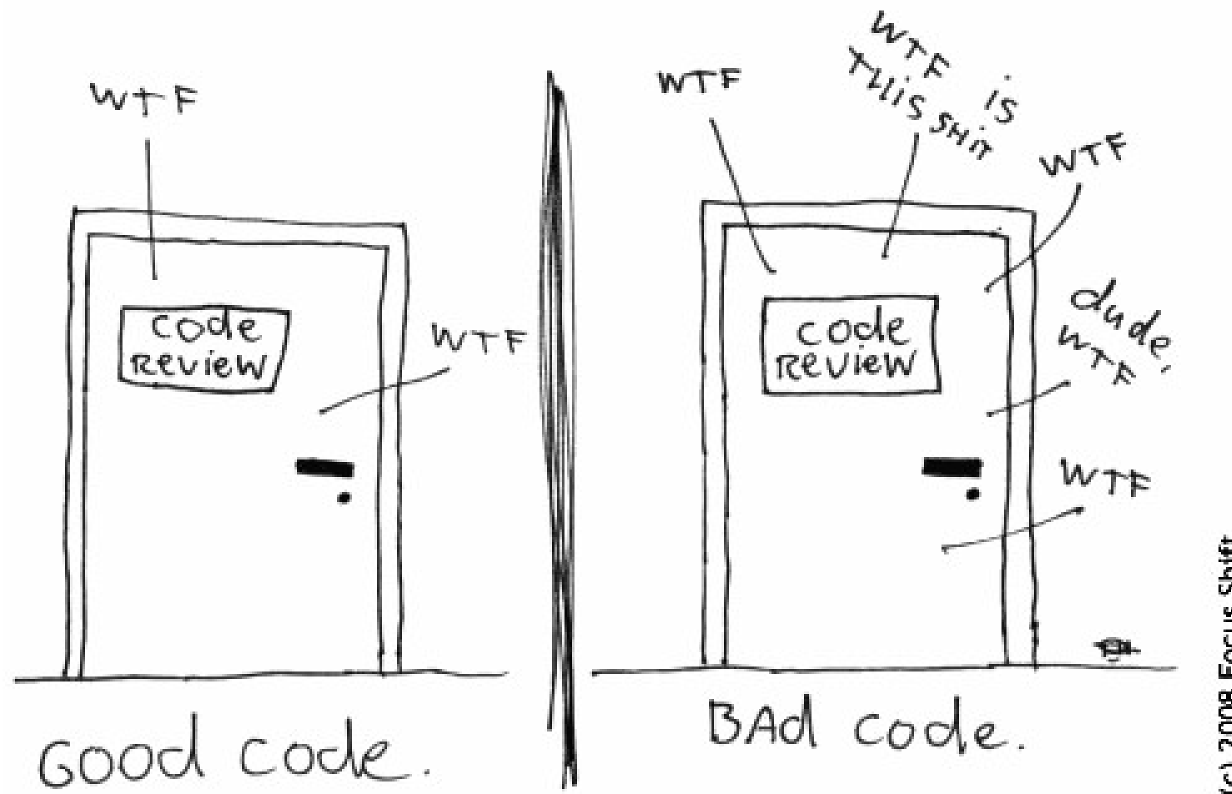
15.11.2022

Why Coding Standards are Important

- “Any fool can write code that a computer can understand. Good programmers write code that humans can understand.” -- *Martin Fowler, "Refactoring: Improving the Design of Existing Code"*
- Improve the readability of code by providing a consistent level of quality
- Code is easier to understand, develop and maintain
- Transition of code to other developers for further maintenance and enhancement is easy

  → Hardly any software is maintained for its whole life by the original author
- Reduce overall costs of the application

Code Quality



Source File Organization

General Code Organization

- One file per top-level class
  - File name equals class name
- Files are encoded in UTF-8
  - Only non-printable characters in Unicode escapes `\Uxxxx`
  - Avoid Unicode escape outside of string literals
- Use spaces for indentation instead of tabs

Source File Structure

- Package statement
- Import statements
  - No wildcard imports
- Class declaration
  - Static fields
    - constants
    - non-constants
  - Non-static fields
  - Constructors
  - Methods
    - setters and getters
    - other methods

Code Formatting

Brackets

- Curly brackets in Kernighan and Ritchie style, aka Egyptian style
  - ➢ No line break before opening brace
  - ➢ Line break after opening brace
  - ➢ Line break before closing brace
  - ➢ Line break after closing brace if brace terminates statement or method body → no line break if **else** or **catch** follows

```
if (condition()) {
      try {
        something();
      } catch (ProblemException e) {
        recover();
      }
    } else if (otherCondition()) {
      somethingElse();
    } else {
      lastThing();
    }
```

Brackets

- Put blocks in curly brackets, even if they only contain a single statement

```
if (a.equals(b)) {
   c = b;
}
```

- Specify the order of operations using round brackets, even if redundant

```
(a && b) || c
```

Indentation and Lines

- Indent your code (!)
- 2 spaces per indentation level
- One statement per line
- A line of code is limited to 100 characters
- If you have to break a line
  - ➢ break *before* non-assignment operator
  - ➢ break *after* comma or assignment operator
  - ➢ Indent continuation lines with 4 spaces per level

```
numberOfParticipants =
    numberOfStudents
        + numberOfTeachers;
```

- Use whitespaces in your code

```
grandTotal = invoice.total() + getAmountDue();
grandTotal=invoice.total()+getAmountDue();
```

- Use single empty lines to organize the code into logical paragraphs

Method Size

- Follow the Thirty-Seconds Rule:
  - Another programmer should be able to fully understand what a method does, why it does it, and how it does it in less than 30 seconds

- If a method is longer than a screen then it is probably too long

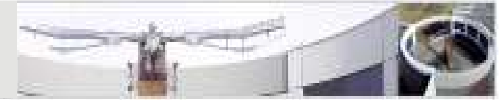
Naming Conventions

What Makes Up a Good Name

- Use mixed case to make names readable
  - lower letters in general
  - capitalize first letter of class and interface names
  - capitalize first letter of non-initial words → CamelCase
  - e.g. `StringTokenizer`

- Use full English descriptors that accurately describe the variable/field/class
  - `firstName, totalSum`
  - `x1, x2, fn`
  - The name is already the first part of the documentation!

What Makes Up a Good Name



- Do not abbreviate names by removing vowels
  - `appendSignature(String signature)`
  - `appndSgntr(String sgntr)`

- Capitalize only the first letter in acronyms
  - `loadXmlDocument()`
  - `loadXMLDocument()`

- Avoid names that are similar or differ only in case
  - `sqlDataBase` vs `sqlDatabase`

Naming Packages, Classes and Interfaces

- Use the reversed, lowercase form of the Internet domain name as root qualifier for package names
  - `de.dfki.mlt.<project>.<subpackage>`

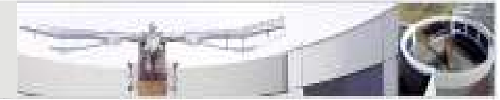
- Use nouns to names classes
  - nouns define objects or *things*
  - `class CustomerAccount { ...`

- Use nouns or adjectives for interfaces
  - `public interface ActionListener { ...`
  - adjectives describe the capability of the implementing class
  - `public interface Runnable { ...`

Naming Methods

- Use a strong, active verb for the first word of a method
  - `openAccount(), printMailingLabel()`
- Getters
  - return the value of a field
  - prefix the word ‘get’ to the name of the field
  - if it is a boolean field, prefix ‘is’ to the name of the field
  - `getFirstName(), isPersistent()`
- Setters
  - modify the values of a field
  - prefix the word ‘set’ to the name of the field
  - `setFirstName(String firstName)`
  - `setPersistent(boolean flag)`

Naming Variables



- Use nouns to name variables
- Pluralize the names of collection references such as arrays and lists
  - `Customer[] customers = ...`
  - Alternative: a suffix like **`Set`** or **`List`**
- Standard names for variables
  - Loop counters: **i, j, k**
  - Exceptions: **e**
- The shorter the name of a variable, the smaller its scope

Naming Fields and Parameters

- Qualify fields with `this.` to distinguish them from local variables

- When a constructor or setter assigns a parameter to a field, give that parameter the same name as the field
  - ```
    private String name;
    public Person(String name) {
      this.name = name;
    }
    ```
  - This is the only situation where name shadowing should occur!
    ```

Naming Constants

- Implemented as static final fields

- Use full English words, all in uppercase, with underscores between the words → UPPER_SNAKE_CASE
  - `MINIMUM_BALANCE, MAX_VALUE`

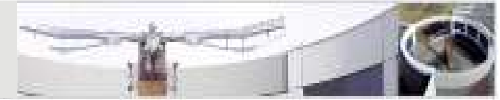
Documentation Conventions

“If your program isn't worth documenting, it probably isn't worth running.” – *Jonathan Nagler, "Coding Style and Good Computing Practices"*

Comment Types

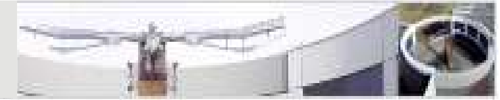
- *Documentation comments* describe the programming interface
  - ```
    /**
     * This is a documentation comment.
     */
    ```
- *Standard comments* hide code without removing it
  - ```
    /*
     This is a standard comment.
     */
    ```
- *One-line comments* explain implementation details
  - ```
    // This is a one-line comment.
    ```
    ```

General Documentation Guidelines



- Comments should add to the clarity of your code
- Keep comments simple
- Keep comments and code in sync
- Write the comments before you write the code
  - ➢ at least comment your code as you write it!
- Write your comments in English!

A Quick Overview of javadoc



- @param <name> <description>
  - Used for methods and constructors
  - Describes the usage of a passed parameter
  - Declare what happens with extreme values (null etc.)
  - Use one tag per parameter

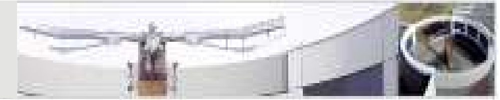
- @return <description>
  - Used for methods
  - Describes the return value, if any, of a method
  - Indicate the potential use(s) of the return value

A Quick Overview of javadoc

- @throws <name> <description>
  - Used for methods and constructors
  - Describes under what circumstances the exception is thrown
  - Use one tag per exception

- @author <name>
  - Used for interfaces and classes
  - Indicates the author(s) of the code
  - Use one tag per author

A Quick Overview of javadoc



- {@link <ClassName#MethodName>}
  - Used for any javadoc comment
  - Generates a hypertext link in the documentation to the specified class or method

- {@code <text>}
  - Used for any javadoc comment
  - Text is displayed verbatim in a fixed-width font
  - Indicates that the text refers to source code

- Standard HTML tags are allowed: <p>, <ul>, <pre>, …

A Quick Overview of javadoc

```
/**
 * Returns a new string that is a substring of this string. The substring
 * begins with the character at the specified index and extends to the
 * end of this string.
 *
 * <p>Examples:
 *
 * <pre>
 * "unhappy".substring(2) returns "happy"
 * "Harbison".substring(3) returns "bison"
 * "emptiness".substring(9) returns "" (an empty string)
 * </pre>
 *
 * @param beginIndex the beginning index, inclusive
 * @return the specified substring, the empty string on border cases,
 *     never returns {@code null}
 * @throws IndexOutOfBoundsException if {@code beginIndex} is negative or
 *     larger than the length of this {@link String} object
 */
public String substring(int beginIndex) {...}
```

Documenting Class Headers

- The purpose of the class
- Known bugs or restrictions
- Author using the appropriate javadoc tag **`@author`**

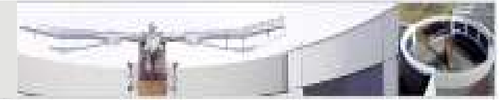
Documenting Method Headers

- What and why the method does what it does

- How a method changes the object or its parameter with side effects

- Document parameters, return value and possible exceptions using the appropriate javadoc tags **`@param, @return`** and **`@throws`**

Documenting Method Bodies



- Rule of thumb: if your code isn't obvious, then you need to document it

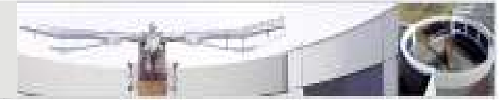
- Document why something is being done, not just what
  - ```
    // increase count by one
    count++;
    ```

- Avoid the use of end-line comments

- Document empty blocks
    ```

Programming Conventions

Variables

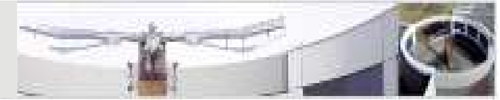


- One variable per declaration
  - `String a, b, c = null;`
- Declare local variables immediately before their use
- Use local variables for one thing only
- Use interfaces for variable types instead of implementing classes if possible
  - e.g. **`Set`** instead of **`HashSet`**, **`List`** or **`Collection`** instead of **`ArrayList`**
  - `Set<String> set = new HashSet<>()`
  - `ArrayList<String> list = new ArrayList<>()`
  - more flexible when replacing the implementation
  - the same is true for the parameters and return types of methods

Class Visibility

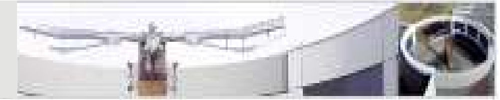
- Use default visibility for classes internal to a component
- Use public visibility for the facades of components

Method Visibility



- Be as restrictive as possible!
- If a method doesn’t have to be public, make it protected
- If a method doesn’t have to be protected, make it private or default
- Minimize the public and protected ‘interface’
  - Improved learnability
  - Reduced coupling

Field Visibility



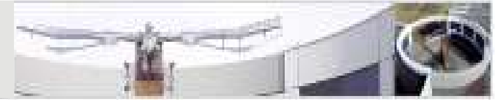
- All non-constant field should be declared *private*
- Ideal Case: The only methods that are allowed to directly work with a field are the getter/setter methods
  - Fields are encapsulated
  - Complete control over how fields are accessed and by whom
  - Enables lazy initialization
  - Handling of side effects
- Relaxation: Define getter/setter for fields that have to be accessed/modified from external classes
  - Internal methods may access fields directly
  - Use the prefix `this.` to distinguish between local variables and fields

- Use unchecked runtime exceptions to indicate errors in your program's logic that cannot be reasonably recovered from at runtime
  - avoid catching runtime exceptions
  - e.g. **`NullPointerException`**

- Use checked exceptions to indicate invalid conditions in areas outside the immediate control of the method
  - e.g. **`IOException`**

- Don’t do `catch (Exception e)` because this also includes runtime exceptions
- Don’t do `throws Exception` because it forces the client to do a `catch (Exception e)`
- Don’t use empty catch blocks
  - at least, add a comment why it’s empty
- Never ever do

```
catch (Exception e) {
}
```



- There is not one ultimate style guide for Java
- Our style guide is largely based on Google’s style guide
- For this course, follow the style guide presented in these slides

- Google Java Style Guide
  - https://google.github.io/styleguide/javaguide.html
- How To Write Unmaintainable Code
  - http://mindprod.com/jgloss/unmain.html
- Robert C. Martin: Clean Code - A Handbook of Agile Software Craftsmanship
  - http://www.amazon.de/Clean-Code-Handbook-Software-Craftsmanship/dp/0132350882/ref=sr_1_1?ie=UTF8&qid=1351073737&sr=8-1