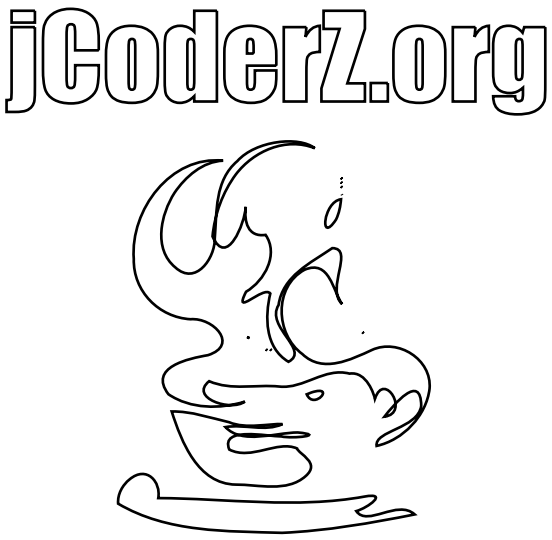
The jCoderZ.org Project
Java Coding Guidelines

Version 1.0



The jCoderZ.org Project Java Coding Guidelines: Version 1.0

Important Notice

Document Control. This document is issued by jCoderZ.org Any queries, or suggestions for improvement to this document should be addressed at our web page http://www.joderz.org/.

Table of Contents

Chapter 1. Introduction

> Science is what we understand well enough to explain to a computer. Art is everything else we do.
>
> —Donald Knuth, *Foreward to the book A=B*

1.1. Why Have Code Conventions

Code conventions are important to programmers for a number of reasons:

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

For the conventions to work, every person at jCoderZ.org writing software must conform to the code conventions.

1.2. Acknowledgements

This document is the result of the merger of 2 sources:

- Java Code Conventions from Sun. The original can be found at http://java.sun.com/docs/codeconv/ [JavaCodeConv].
- The former code conventions we where used to adhere.

The document is maintained by the jCoderZ.org Project. Comments can be sent the web site at http://www.jcoderz.org/.

1.3. Document content

This document contains a list of rules ordered by different aspects. In general all given rules must be respected in order to make the code more readable.

If conforming to these rules leads to code that is harder to read and understand, you might break these rules and choose another style for small parts of your code. You must prefer open standards when using another convention and document the convention and the reason for breaking these rules.

The inability to conform to this guideline might be a strong indication that the given code approach could be enhanced. To avoid major rework tasks, please get in contact with any member of the jCoderZ.org Project in order to get assistance for the given coding problem.

If you are faced with a problem that is not covered by any of the rules in this document, get in contact with the jCoderZ.org Project or use the referred document to identify a solution.

The Book *Writing Robust Java Code* [Ambler00] is a reading recommendation and gives a good introduction for novice and advanced software engineers.

Conformance to this coding guideline will automatically be checked by Checkstyle. If non-conforming code is detected, the code author will be contacted and asked to enhance his source code.

Chapter 2. Source File Structure

A file consists of sections that should be separated by blank lines and an optional comment identifying each section. Files longer than 2000 lines are cumbersome and should be avoided.

Each Java source file must contain a single toplevel class or interface.

Java source files must have the following ordering:

- File Header
- Package and Import statements
- Class and interface declarations

2.1. File Header

All source files must begin with the following header:

```
/*
 * $Id: SampleSnippets.java 1011 2008-06-16 17:57:36Z amandel $
 *
 * Copyright 2006, The jCoderZ.org Project. All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions are
 * met:
 *
 *    * Redistributions of source code must retain the above copyright
 *      notice, this list of conditions and the following disclaimer.
 *    * Redistributions in binary form must reproduce the above
 *      copyright notice, this list of conditions and the following
 *      disclaimer in the documentation and/or other materials
 *      provided with the distribution.
 *    * Neither the name of the jCoderZ.org Project nor the names of
 *      its contributors may be used to endorse or promote products
 *      derived from this software without specific prior written
 *      permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
 * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS AND CONTRIBUTORS
 * BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
 * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
 * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
 * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
 * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
 * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
 * ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
 */
```

You don't have to bother with filling in Id into this header as they are automatically filled via CVS/SVN functionality.

2.2. Package and Import statements

The package declaration must follow immediately after the file header.

For import statements the “single-type import statements” must always be used:

```
import java.io.IOException;
import java.io.Serializable;
```

Using single-type imports is quite useful and makes it easy for the reader to determine the package of a particular type. Do not use “On-demand import statements”:

```
import java.util.*; // DON'T
```

Import statements should be lexicographically sorted and grouped according to the upper level packages (Recommendation).

2.3. Class and Interface Declarations

The following table describes the parts of a class or interface declaration, in the order that they should appear:

Table 2.1. Class/Interface parts order

No.	Part of Class/Interface	Declaration Notes
1	Class/interface documentation comment (`/**...*/`)	See Documentation Comments for information on what should be in this comment.
2	Class or interface statement	
3	Class/interface implementation comment (`/*...*/`), if necessary.	This comment should contain any class-wide or interface-wide information that wasn't appropriate for the class/interface documentation comment.
4	Class (static) variables [a]	
5	Instance variables [a]	
6	Constructors [a]	
7	Methods [a]	

[a] Order is always from public to most private: public, protected, then package level (no access modifier), private.

Chapter 3. Comments

3.1. Implementation Comments

Implementation comments are those which are delimited by `/*...*/` and `//`. They are means for commenting out code or for adding information about the particular implementation. Doc comments are meant to describe the specification of the code, from an implementation-independent perspective. It is meant for developers who might not necessarily have the source code at hand.

Comments should be used to give an overview of the code and provide additional information that are not readily available in the code itself. Comments should contain only information that are relevant to reading and understanding the program. For example, information about how to build the corresponding package or in what directory it resides should not be stated as a comment.

Discussions of non-trivial or non-obvious design decisions are appropriate, but avoid duplicating information that is present and clearly visible in the code. Redundant comments get outdated too easily. In general, avoid any comments that are likely to get outdated as the code evolves.

Note: A high frequency of comments sometimes reflects poor quality of code. When you feel compelled to add a comment, consider rewriting the code to make it clearer. Comments must not be enclosed in large boxes drawn with asterisks or other characters.

3.2. Documentation Comments

Doc comments describe Java classes, interfaces, constructors, methods, and fields. Each doc comment is set inside the comment delimiters `/**...*/`.

```
/**
 * The Example class provides ...
 * @author Stephen Mohr
 * @author Oliver Griffin
 */
public class ExampleClass
{
    // ...
}
```

The first line of doc comments (`/**`) for classes and interfaces is not indented. Subsequent doc comment lines each have 1 space of indentation (to vertically align the asterisks). Members, including constructors, have three spaces for the first doc comment line and four spaces thereafter. Doc comments should not be positioned inside a method or constructor definition block, because Java associates documentation comments with the first declaration after the comment.

Doc comments for classes or interfaces must include the `@author` tag. Only one name per @author tag in the form `@author` *firstname lastname*. Developers making major changes on the file must add their name. For methods or constructors the `@param`, `@return`, `@throws`, `@see` tag must be included as needed. Do not use the `@exception` tag. A `package.html` must be added for each new package.

For further details, see How to Write Doc Comments for Javadoc which includes information on the doc comment tags and for details about doc comments and javadoc, see the Javadoc Tool Home Page.

Chapter 4. Statements

4.1. Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces `{ statements }`. See the following sections for examples.

- The enclosed statements must be indented one more level than the compound statement.
- The opening and the closing brace must begin in a new line and be indented to the beginning of the compound statement.
- Braces are used around all statements, even single statements, when they are part of a control structure, such as an if-else or for statement. This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

4.2. return Statements

A return statement with a value should not use parentheses unless they make the `return` value more obvious/better readable in some way.

```
return list.size();
// OR
return (size != 0 ? size : DEFAULT_SIZE);
```

4.3. if, if-else, if else-if else Statements

```
if (i > 0)
{
    // ...
}

if (i > 0)
{
    // ...
}
else
{
    // ...
}

if (i > 0)
{
    // ...
}
else if (i == 0)
{
    // ...
}
else
{
    // ...
}
```

4.4. for Statements

```
for (int j = 0; j < SIZE; j++)
{
    // ...
}
```

When using the comma operator in the initialization or update clause of a `for` statement, don't use more than three variables. If needed, use separate state-

ments before the `for` loop (for the initialization clause) or at the end of the loop (for the update clause).

4.5. while Statements

```
while (i > 0)
{
    // ...
}
```

Don't do any operations within the control element:

```
while (--i > 0);  // DON'T
```

4.6. do-while Statements

```
do
{
    // ...
}
while (i > 0);
```

4.7. switch Statements

```
switch (i)
{
    case HttpServletResponse.SC_ACCEPTED:
        // ...
        /* falls through */
    case HttpServletResponse.SC_BAD_REQUEST:
        // ...
        break;
    case HttpServletResponse.SC_CONTINUE:
        // ...
        break;
    default:
        throw new RuntimeException("Unexpected condition.");
        // no break here because position is unreachable!
}
```

Every time a case falls through (doesn't include a break statement), add a comment where the break statement would normally be to indicate that the fall-through is happening intentionally. This is shown in the preceding code example with the `/* falls through */` comment.

Every `switch` statement must include a default `case`. The `break` in the `default` case is redundant, but it prevents a fall-through error if later another case is added. The `default` case should anyway always be the last to appear.

4.8. try-catch Statements

```
try
{
    // ...
}
catch (IllegalArgumentException ex)
{
    // ...
}

try
{
    // ...
}
```

```
catch (IllegalArgumentException ex)
{
    // ...
}
finally
{
    // ...
}

try
{
    // ...
}
finally
{
    // ...
}
```

Chapter 5. Whitespace and Indentation

5.1. Indention

Four spaces should be used as the unit of indentation. Never use tabs. All indentation must be done using space characters.

Do not write lines longer than 80 characters, since they are not handled well by many terminals and tools.

Note: Examples for use in documentation should have a shorter line length (no more than 70 characters).

Break always before the keywords `extends`, `implements` and `throws` and indent a additional unit of indention (4 spaces).

```
public class IndentionSample
    extends SampleSnippets
    implements Serializable, Cloneable, Comparable
{
    /**
     *
     */
    public void doSomething (int length)
        throws IOException
    // ...
```

When an expression will not fit on a single line, break it according to these general principles:

- Break field and variable initializers before the '=' operator.
- Break after opening parenthesis, comma or dot.
- Break before an operator or closing parenthesis.
- Prefer higher-level breaks to lower-level breaks.
- Base indent is two additional units of indention (8 spaces). Deeper nested expressions are further indented according to their nesting level. (12, 16, ... spaces)

```
final SimpleBusinessResultException e
        = new SimpleBusinessResultException(
            ResultCode.SPLIT_AUTHORIZATION_SPLIT_INDEX_UNEXPECTED);
```

If the above rules lead to confusing code or to code that is squished up against the right margin, just indent 8 spaces instead.

If your code is so deeply nested, splitting the code into several methods might be a good idea. Also if your statement is much too long to fit in a line you might think about rewriting the code using several statements.

5.2. Blank Lines

Blank lines improve readability by setting off sections of code that are logically related.

One blank line should always be used in the following circumstances:

- Between methods
- Before a block or single-line comment

- Between logical sections inside a method to improve readability
- Between groups of import statements

Two blank lines should always be used between sections of a source file.

5.3. Blank Spaces

Blank spaces must be used in the following circumstances:

- A keyword followed by a parenthesis must be separated by a space.

```
while (i > 0)
{
    // ...
}
```

  Note that a blank space must not be used between a method name and its opening parenthesis, except at the method declaration (see below). This helps to distinguish keywords from method calls.
- A blank space must appear after commas in argument lists.
- All binary operators except `.` must be separated from their operands by spaces. Blank spaces must not separate unary operators such as unary minus, increment ++, and decrement `--` from their operands.

```
int d = 1;
b = (a + b) / (c * ++d);
System.out.println("c=" + c + "\n");
```

- The expressions in a `for` statement must be separated by blank spaces.

```
for (int j = 0; j < SIZE; j++)
{
    // ...
}
```

- Casts must be followed by a blank space.

```
final boolean result = aMethod((byte) a, (Object) x);
anotherMethod((int) (a + 1), ((int) (b + MAX_LOOPS)) + 1);
```

- Method/Constructor declarations. To allow easy searching for method or constructor declarations put a whitespace between the method name and the opening paranthesis of the parameter list. When doing a method invocation, do not put a whitespace between method name and paranthesis. This allows to distinguish between method invocation and declaration.

```
private static void anotherMethod (int a, int b)
{
    // ...
}
```

Chapter 6. Naming Conventions

Naming conventions make programs more understandable by making them easier to read. They can also give information about the function of the identifier - for example, whether it's a constant, package, or class - which can be helpful in understanding the code.

If acronyms or abbreviations are used in a name only the first letter might be uppercase (except for Constants). So choose class `HtmlGateway` not class `HTMLGateway`.

6.1. Packages

The prefix of a unique package name is always `org.jcoderz.` and must match the regular expression `^org\.jcoderz(\.[a-z][a-z0-9]*)+$`.

Subsequent components of the package name vary according to the teams own internal naming conventions.

6.2. Classes

Class names should be nouns, in mixed case with the first letter of each internal word capitalized and must match the regular expression `^[A-Z][a-zA-Z0-9]*$`.

Try to keep your class names simple and descriptive. Use whole words - avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML), e.g. `class Raster` or `class ImageSprite`.

6.3. Interfaces

Use nouns to name interfaces that act as service declaration:

```
public interface ActionListener
{
    void actionPerformed (EventObject event);
}
```

Use adjectives to name interfaces that act as descriptions of capabilities. Most interfaces that describe capabilities use an adjective created by tacking an “able” or “ible” suffix to onto the end of verb:

```
public interface Runnable
{
    void run ();
}

public interface Accessible
{
    Context getContext ();
}
```

Interface names must, like class names, have the first letter (of each noun) capitalized.

6.4. Methods

Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized: `run()`, `runFast()`, or `getBackground()`.

6.5. Variables

Variables are in mixed case with a lowercase first letter, internal words start with capital letters. Variable names must not start with underscore _ or dollar sign $ characters.

Variable names should be short yet meaningful. The choice of a variable name should be mnemonic - that is, designed to indicate to the casual observer the intent of its use. One-character variable names must be avoided except for temporary throwaway variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.

Do not use local variable names that hide variables at higher levels.

The name of class members must start with a lowercase letter `s` and match the regular expression `^s[A-Z][a-zA-Z0-9]*$`. The name of natural members must start with a lowercase `m` and match the regular expression `^m[A-Z][a-zA-Z0-9]*$`.

```
public class MemberSample
{
    private static int sClassAccessCounter = 0;
    private int mMemberAccessCounter = 0;

    // ...
}
```

Names of variables that refer to collections of objects should correspond to the plural form of the semantic type contained in the collection. This enables a reader of the code to distinguish between variables representing multiple values from those representing single values:

```
private Object[] mCustomers = new Object[MAX_CUSTOMERS];

void addCustomer (int index, Object customer)
{
    mCustomers[index] = customer;
}
```

6.6. Constants

The names of constants must be all uppercase with words separated by underscores (_). Exception to this are the constants `logger` and `serialVersionUID`.

```
static final int MIN_WIDTH = 4;
static final int MAX_WIDTH = 999;
static final int GET_THE_CPU = 1;
static final long serialVersionUID = -7064645359225861305L;
static final Logger logger
        = Logger.getLogger(SampleSnippets.class.getName());
```

Chapter 7. Programming Practices

7.1. Referring to Class Variables and Methods

Do not use an object to access a class (static) variable or method. Use a class name instead.

```
classMethod();
ReferringSample.classMethod();
anObject.classMethod(); // DON'T
```

7.2. Constants

Numerical constants (literals) must not be coded directly (magic numbers), except for -1, 0, and 1, which can appear in a `for` loop as counter values.

7.3. Parenthesis

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to you, it might not be to others. Do not assume that other programmers know precedence as well as you do.

7.4. Returning Values

You should have only one exit point in a method. You must have a good explanation if you use more than one return statement in a method.

7.5. Special Comments

Use `TODO` in a comment to flag something that is bogus but works. Use `FIXME` to flag something that is bogus and broken.

7.6. Initialization

Initialize local variables where they are declared. The only reason not to initialize a variable in its declaration is if the initial value depends on some computation that has to occur first.

7.7. Size Contraints

Methods are limited to 100 lines of code. Empty lines and single line comments are ignored.

The number of arguments for an method or constructor must not exceed 7.

7.8. Empty Blocks

Intentionally empty block must contain a comment. Empty blocks that can/should never be reached like empty catch or default block must throw a RuntimeException.

Chapter 8. Java Source File Examples

The following example shows how to format a Java source file containing a single public class. Interfaces are formatted similarly.

```
 1 /*
 2  * $Id: TransactionId.java 1011 2008-06-16 17:57:36Z amandel $
 3  *
 4  * Copyright 2006, The jCoderZ.org Project. All rights reserved.
 5  *
 6  * Redistribution and use in source and binary forms, with or without
 7  * modification, are permitted provided that the following conditions are
 8  * met:
 9  *
10  *    * Redistributions of source code must retain the above copyright
11  *      notice, this list of conditions and the following disclaimer.
12  *    * Redistributions in binary form must reproduce the above
13  *      copyright notice, this list of conditions and the following
14  *      disclaimer in the documentation and/or other materials
15  *      provided with the distribution.
16  *    * Neither the name of the jCoderZ.org Project nor the names of
17  *      its contributors may be used to endorse or promote products
18  *      derived from this software without specific prior written
19  *      permission.
20  *
21  * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS "AS IS" AND
22  * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
23  * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
24  * PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS AND CONTRIBUTORS
25  * BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
26  * CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
27  * SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR
28  * BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY,
29  * WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR
30  * OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF
31  * ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
32  */
33 package org.jcoderz.guidelines.snippets;
34
35
36 import java.io.File;
37 import java.io.FileInputStream;
38 import java.io.IOException;
39 import java.io.Serializable;
40
41
42 /**
43  * This util class represents an Transaction Id with all
44  * of its features and restrictions.
45  * Instances of this class are immutable.
46  *
47  * @author SWAG
48  */
49 public final class TransactionId
50     implements Comparable, Serializable
51 {
52     /** Name of this type. */
53     public static final String TYPE_NAME = "TX_ID";
54
55     /** Bit mask used for hashcode generation. */
56     private static final int NUMBER_OF_BITS_PER_INT = 32;
57
58     private static final int BUFFER_SIZE = 4096;
59
60     private static final int BUFFER_MULTIPLIER = 2;
61
62     private static final long serialVersionUID = -7064645359225861305L;
63
64     /** Holds the transaction id */
```

```
 65     private final long mTransactionId;
 66
 67
 68     /**
 69      * Creates a new instance of TransactionId.
 70      *
 71      * @param transactionId the transaction to be represented by the
 72      *       <code>TransactionId</code>.
 73      * @throws IllegalArgumentException if the long does not fit
 74      *       into a transaction id.
 75      */
 76     private TransactionId (long transactionId)
 77         throws IllegalArgumentException
 78     {
 79         if (transactionId < 0)
 80         {
 81             throw new IllegalArgumentException(TYPE_NAME + " "
 82                     + String.valueOf(transactionId)
 83                     + "Value must be positive.");
 84         }
 85         mTransactionId = transactionId;
 86     }
 87
 88
 89     /**
 90      * Parses the string argument as a transaction id.
 91      *
 92      * @param s the <code>String</code> containing the transaction id.
 93      * @return the transaction id represented by the string argument.
 94      * @throws IllegalArgumentException if the string does not contain a
 95      *       parseable transaction id.
 96      */
 97     public static TransactionId fromString (String s)
 98         throws IllegalArgumentException
 99     {
100         final TransactionId result;
101         try
102         {
103             result = new TransactionId(Long.parseLong(s));
104         }
105         catch (NumberFormatException ex)
106         {
107             final IllegalArgumentException iaex
108                     = new IllegalArgumentException(
109                         TYPE_NAME + " Failed to parse the value.");
110             iaex.initCause(ex);
111             throw iaex;
112         }
113         catch (NullPointerException ex)
114         {
115             final IllegalArgumentException iaex
116                     = new IllegalArgumentException(
117                         TYPE_NAME + " Value must not be null.");
118             iaex.initCause(ex);
119             throw iaex;
120         }
121         return result;
122     }
123
124     /**
125      * Returns a transaction id from the given long <code>l</code>.
126      *
127      * @param l the <code>long</code> containing the transaction id.
128      * @return the transaction id represented by the argument.
129      * @throws IllegalArgumentException if the long does not fit into a
130      *       transaction id.
131      */
132     public static TransactionId fromLong (long l)
133         throws IllegalArgumentException
134     {
135         return new TransactionId(l);
136     }
```

```
137
138     /**
139      * Returns the transaction id as String.
140      *
141      * @return the transaction id as String.
142      */
143     public String toString ()
144     {
145         return Long.toString(mTransactionId);
146     }
147
148     /**
149      * Returns the transaction id as long.
150      *
151      * @return the transaction id as long.
152      */
153     public long toLong ()
154     {
155         return mTransactionId;
156     }
157
158     /**
159      * Indicates whether some other object is "equal to" this one.
160      *
161      * @param obj the object to compare this <code>TransactionId</code>
162      *       against.
163      * @return true if this object is the same as the obj argument; false
164      *       otherwise.
165      */
166     public boolean equals (Object obj)
167     {
168         boolean result = false;
169
170         if (obj instanceof TransactionId)
171         {
172             result = (mTransactionId
173                     == (((TransactionId) obj).mTransactionId));
174         }
175
176         return result;
177     }
178
179     /**
180      * Compare two transaction IDs.
181      * This implementation is consistent with {@link #equals(Object)}.
182      *
183      * @param o object with which to compare this TransactionId
184      * @return a result less than zero if this object is less than
185      *       <code>o</code>, exactly zero if they are equal, and a result
186      *       greater than zero otherwise.
187      */
188     public int compareTo (Object o)
189     {
190         final int result;
191         // Can't simply return the difference, because that difference
192         // might not fit in an int.
193         if (mTransactionId < ((TransactionId) o).mTransactionId)
194         {
195             result = -1;
196         }
197         else if (mTransactionId > ((TransactionId) o).mTransactionId)
198         {
199             result = 1;
200         }
201         else
202         {
203             result = 0;
204         }
205         return result;
206     }
207
208     /**
```

```
209      * Compute hash code.
210      *
211      * @return hash code for this transaction ID
212      */
213     public int hashCode ()
214     {
215         return (int) (mTransactionId
216                 ^ (mTransactionId >>> NUMBER_OF_BITS_PER_INT));
217     }
218
219     /**
220      * Helper function to read the full content of the file.
221      *
222      * @param file the file to read.
223      * @return the content of the given file as byte array.
224      * @throws IOException if a IOException occurs.
225      */
226     private static byte[] readFully (File file)
227         throws IOException
228     {
229         final FileInputStream in = new FileInputStream(file);
230         byte[] buffer = new byte[BUFFER_SIZE];
231         int read;
232         int pos = 0;
233
234         while ((read = in.read(buffer, pos, buffer.length - pos)) > 0)
235         {
236             pos += read;
237             if (pos == buffer.length)
238             {
239                 byte[] newBuffer
240                 = new byte[buffer.length * BUFFER_MULTIPLIER];
241
242                 System.arraycopy(buffer, 0, newBuffer, 0, buffer.length);
243                 buffer = newBuffer;
244             }
245         }
246
247         if (pos != buffer.length)
248         {
249             byte[] newBuffer = new byte[pos];
250
251             System.arraycopy(buffer, 0, newBuffer, 0, pos);
252             buffer = newBuffer;
253         }
254
255         return buffer;
256     }
257 }
258
```

Appendix A. Appendix

References

[Ambler00] *Writing Robust Java Code*. Scott W. Ambler. Copyright © 1998, 1999 AmbySoft Inc.. http://www.ambysoft.com/javaCodingStandards.pdf .

[bloch01] *Effective Java Programming Language*. Joshua Bloch. Copyright © 2001 Addison-Wesley Professional. Addison-Wesley Professional . 0201310058.

[JavadocHowTo] *How to Write Doc Comments for the JavadocTM Tool*. Copyright © 2000 Sun Microsystems, Inc. http://java.sun.com/j2se/javadoc/writingdoccomments/index.html .

[SunCodeConv] *Code Conventions for the Java Programming Language*. Copyright © 1995, 2003 Sun Microsystems, Inc. http://java.sun.com/docs/codeconv/ .

[checkstyle] *Checkstyle*. http://checkstyle.sourceforge.net/ .

[pmd] *PMD*. http://pmd.sourceforge.net/ .

[findbugs] *FindBugs - A Bug Pattern Detector for Java*. http://www.cs.umd.edu/~pugh/java/bugs/ .