

Software Development (CS2500)

Lecture 14: Javadoc and Coding Conventions

M.R.C. van Dongen

November 3, 2010

Contents

1	Introduction	1
2	javadoc	2
3	Coding Conventions	4
3.1	Files	5
3.2	Classes and Interfaces	5
3.3	Indentation	6
3.4	Comments	7
3.5	Declarations	8
3.6	Statements	8
3.7	White Space	10
3.8	Naming Conventions	11
3.9	Methods	12
3.10	Other Practice	12
4	Acknowledgements	12
5	For Friday	12

1 Introduction

This lecture studies the javadoc documentation mechanism, and some important Java coding conventions. From now on you are expected to use javadoc and adhere to the coding conventions for all *future* assignments.

2 Generating Documentation

The javadoc tool automates the writing of documentation from Java programs. The documentation is generated from comments in java programs. The comments are formatted in a special style called *doc comments*. To create the documentation you run the javadoc program on the input .java file.

```
$ javadoc LovelyClass.java
```

Unix Session

Doc comments start with `/**` and end in `*/`. Additional lines should start with `*`. The comments may contain html tags.

```
/**
 * This is an <strong>example</strong>.
 */
```

Java

The first line of each doc is automatically included in the resulting documentation. It should provide a concise description of what is documented. Doc comments are subdivided into *descriptions* and *tags*.

Description: Descriptions provide an overview of the functionality of the presented code.

Tag: Tags specify/address specific information. This includes information about author, version, and so on.

Tags are used to specify content and markup. Tags are case-sensitive and should start with `@`.

```
/**
 * Basic print method.
 *
 * @author Java Joe.
 * @param bar the thing to be printed.
 */
public void printStuff( int bar ) {
    :
}
```

Java

There are two kinds of tags: *block tags* and *inline tags*. *Block tags* are of the form `@(tag name)`. Block tags should be placed in the tag section following the main description. *Inline tags* are of the form `{@(tag name) <more>}`. They may occur anywhere.

```

/**
 * Friendly class.
 * More information {@link #hello here}.
 */
public class Hello {
    public static void hello( ) {
        System.out.println( "hello world." );
    }
}

```

The following are some existing tags. More information may be found at <http://sun.com/j2se/javadoc/>.

@author: Author Entry. Usually there is an author entry for each author in chronological order from top to bottom, but you may have several authors per @author line. Each @author entry should have a name or a comma-separated list of names. The @author tags are only used if you run javadoc with the -author option.

@param: Parameter Entry. A @param entry is required for each parameter. Each @param should be followed by the name of the parameter and a one-line description of the purpose of the parameter. The entries are usually listed from top to bottom in the same order as the formal parameter list. A parameter entry may also describe a *generic type* parameter. These are written in angular brackets. You will learn about generic types in some future lecture.

@version: Version Entry. This should be followed by a <version text>. The <version text> is a free format description. If javadoc is used with the -version option then the resulting documentation will have a subheading describing the <version text>. There is no need to include @version entries in your assignment programs.

@return: Return Entry. This should be followed by a description of the return value.

The following is an example.

```

/**
 * Compute length of a given list.
 *
 * @param list The given list.
 * @param <T> The type of the elements in the list.
 * @return The length of the list.
 */
public int length( List<T> list ) ...

```

You can also define hyperlinks. The following creates a hyperlink for <text> with destination the method <member> in class <class> and package <package>. It is also possible to omit <package> and <class>.

```
/**
 * {@link <package>.<class>#<member> <text>}
 */
```

It is recommended that you present block tags in the following order. Some of these tags may not be required for methods and others not for classes.

```
/**
 * @param      ...
 * @return     ...
 * @exception   ...
 * @author     ...
 * @version    ...
 * @see        ...
 * @since      ...
 * @serial     ...
 * @deprecated ...
 */
```

3 Coding Conventions

This section studies some coding conventions, most of which have been adopted as standard by Sun. Before we start studying the conventions, let's have a look at why we should care about conventions.

- *80% of the lifetime cost of software goes to maintenance.* Clearly it is easier to maintain code if it adheres to some standard.
- Hardly any software is maintained for its whole lifetime by the original author. By adhering to conventions your code becomes more predictable. If there are no standards then this makes it more difficult to find the necessary information in your code.
- Coding conventions are aimed at improving the readability of your code and making it easier for others to understand your code.
- You need to make sure your shipped code is well packaged and clean.

The remainder of this section is almost completely based on Sun's coding conventions. It has been tried not to refer to any notions which we haven't studied yet.

Most importantly, try to follow the coding conventions of your company and/or the person whose code you're modifying.

3.1 Files

- Files should consist of sections. Sections should be separated by blank lines and comments.
- Files longer than 2000 lines should be avoided.
- Javadoc comments should be used to document the classes/interfaces, attributes, and methods.
- The import statements always go to the top of the file.
- If you need more than one import from a package, use the * notation:

```
import java.util.*;
```

Convention

This is better than the following:

```
import java.util.TreeMap;  
import java.util.Random;
```

Don't Try this at Home

3.2 Classes and Interfaces

An *interface* is a collection of methods without bodies. They will be explained in some future lecture. Class and interface declarations should be organised from top to bottom as follows:

1. If you need one, the package statement comes first. For the moment you may forget about this.
2. Next come the import statements.
3. This is followed by the class-related javadoc comments.
4. Class variables in increasing order of visibility: public, protected, and private.
5. Instance variables in increasing order of visibility.
6. Constructors.
7. Methods.

Implementation comments should be used where appropriate.

Place the braces that start and end the class/interface as follows:

```
class Example { // Opening brace here.  
:  
:  
} // Closing brace here.
```

Convention

3.3 Indentation

The following are the guidelines for code indentation.

- Use four spaces as the unit for indentation.
- Use eight spaces if that improves readability.
- Avoid lines that are longer than 80 characters. Come to think of it, make that 70 characters. The reason for this rule is that long lines are difficult to scan from left to right. In addition, many printers cannot print lines that are longer than ± 74 characters. If you print long lines on such printers the lines will wrap, thereby making it impossible to read your code.
- Use the following rules for wrapping lines if they're too long:
 - Break after a comma;
 - Break before an operator;
 - Prefer higher-level breaks to lower-level breaks; and
 - Align the text on the new line with the broken expression on the previous line.
- Compound statements (blocks):
 - The enclosed statements should be indented one more level.
 - The opening brace should be at the end of the line that begins the compound statement.
 - The closing brace should be indented at the same level as the line on the beginning of the block.

The following are two examples of how methods may be broken according to these rules.

```
call1( longExpr1, longExpr2, longExpr3,  
      longExpr2, longExpr3 );  
  
int var = call2( longExpr1,  
                call3( longExpr2, longExpr3,  
                      longExpr2, longExpr3 ) );
```

Convention

Notice that we could have broken the last call as follows. However, this is not ideal as it does the breaking at a more deeply-nested level.

```
int var = call2( longExpr1, call3( longExpr2,  
                                longExpr3, longExpr2, longExpr3 ) );  
int var = call2( longExpr1, call3( longExpr2,  
                                longExpr3,  
                                longExpr2,  
                                longExpr3 ) );
```

Don't Try this at Home

The following is an example of how to use these rules to break arithmetic expressions.

```
longVariable = longExpr1 + (longExpr2 - longExpr3)
                    / longExpr5;
```

Convention

The following is worse than the previous example because it breaks the expression at a deeper level (inside the parentheses).

```
longVariable = longExpr1 + (longExpr2
                          - longExpr3) / longExpr4;
```

Don't Try this at Home

In the following example, the level of indentation after the breaking is increased by 4 spaces to improve readability. Without it it would have been difficult to read the body of the `if` statement.

```
if ((condition1 && condition2)
    || (condition3 && condition4)) {
    // Stuff
}
```

Convention

Clearly this is better than the following:

```
if ((condition1 && condition2)
    || (condition3 && condition4)) {
    // Stuff
}
```

Don't Try this at Home

The following is a suggestion for breaking the ternary conditional expression:

```
var1 = (condition) ? thisStuff : thatStuff;
var2 = (condition) ? thisStuff
                : thatStuff; // Clearer!
var3 = (condition)
      ? thisStuff
      : thatStuff; // Also impossible to miss!
```

Convention

3.4 Comments

As a general rule, prefer end-of-line comments inside methods. The reason for using them is that block comments don't nest.

```
public int answer( ) {
    /* Temporarily commented out for testing.
    /*
    * This gives you the answer.
    */
    */
    return 42;
}
```

Don't Try this at Home

However, the following does work.

```
public int answer( ) {
    /* Temporarily commented out for testing.
    //
    // This gives you the answer.
    //
    */
    return 42;
}
```

Java

3.5 Declarations

Ideally, there should be one declaration per line.

```
int one; // Comment about purpose of one.
int two; // Comment about purpose of two.
```

Convention

This rule improves readability and improves commenting. Never, ever, mix different types in a declaration.

```
int one, many[]; // Allowed, but don't do this.
```

Don't Try this at Home

Use variable declarations that minimise the scope [Bloch, 2008, Item 29].

3.6 Statements

This section presents the conventions for general statements.

There should be no more than one statement per line. So avoid the following:

```
thisVar ++; thatVar --;
```

Don't Try this at Home

The comma operator allows you to put several statements in a single statement. These statements are separated using commas. Technically speaking, the resulting statement is a single statement. Still it is recommended that you avoid using the comma operator.


```
thisVar ++, thatVar --;
```

Don't Try this at Home

Avoid parentheses for return statements (*unless this makes it clearer*).

```
return; // Allowed but not for this module.  
:  
:  
return myLuvelyComputation( );  
:  
:  
return (condition ? thisValue : thatValue);
```

Convention

Always use braces for if statements in a similar style as the following.

```
if (condition1) {  
    ...  
}  
if (condition2) {  
    ...  
} else {  
    ...  
}  
if (condition3) {  
    ...  
} else if (condition4) {  
    ...  
} ...
```

Convention

For for statements with a non-empty body, always use braces in a similar style as the following (even if there's only one statement).

```
for ( initialisation; condition; update ) {  
    ...  
}
```

Convention

For for statements with an empty body, add a semicolon after the closing parenthesis in a similar style as the following.

```
for ( initialisation; condition; update ) ; // empty body
```

Convention

Arguably it is clearer to use a while loop:

```
initialisation;  
while (condition) {  
    update  
}
```

Java

For while statements with a non-empty body, always use braces in a similar style as the following (even if there's only one statement).

```
while ( condition ) {  
    ...  
}
```

Convention

For while statements with an empty body, add a semicolon after the closing parenthesis in a similar style as the following.

```
while ( condition ) ; // empty body
```

Convention

Arguably it is clearer to use the do-while statement:

```
do {  
} while ( condition );
```

Convention

For the do-while statement always use braces in a similar style as the following

```
do {  
    ...  
} while ( condition );
```

Convention

3.7 White Space

Adding white space generally improves readability. Add a blank line for the following:

- Between method definitions.
- Between local variable declarations at the start of a block and the statements in the block.
- Before a block.
- Between logical sections inside a method to improve readability.

Blank spaces should be used in the following circumstances:

- A keyword followed by a parenthesis:

```
while(condition) {
```

Convention

- A parenthesis followed by a brace:

```
while (condition){
```

Convention

- After commas in argument lists.
- Before and after binary operators (except .):

```
var1 = var2 + var3 * var4 / (var5.method( ) - 1);
```

Convention

- After the semicolons in the for statement:

```
for (start; condition; update) {
```

Convention

- After a cast: '(int)(3 * Math.random())'.

3.8 Naming Conventions

Java does not impose any restriction on identifier names for classes, interfaces, variables, and methods. However, by carefully naming them this makes it easier to recognise their type and purpose in a program. The following are the conventions.

Classes: Class names should be nouns in mixed case. The first letter in each internal word should be upper case. Use whole words and avoid acronyms (unless they're widely accepted such as URL, HTML, and so on.). Acceptable class names are: Dog, CatFood, MouseFoodFactory,

Interfaces: Ideally interfaces should be mixed case adjectives ending in 'able' or 'ible'. Otherwise, their names are similar as class names. Acceptable interface names are: Comparable, Sortable, Incomprehensible,

Methods: Method names should be *meaningful* verbs in mixed case. The first letter should be lower case. The first letter of the remaining internal words should be upper case. Acceptable method names are: compute, addNumbers,

Constants: Class constants should be upper case with words separated with underscores. Acceptable constants names are: MAX_VALUE, MINIMUM,

Variables: Variables should be short, yet meaningful nouns. The naming scheme is the same as for methods. Acceptable variable names are: number, value, keyToLock,

3.9 Methods

Methods should be short. Break methods in sub-method calls when they become longer than, say, 40 lines.

3.10 Other Practice

Other good programming practice will be announced when you're ready for it.

4 Acknowledgements and Further Information

The section about javadoc is partially based on [Lewis and Loftus, 2009, Appendix I]. More information about javadoc may be found at the following URL: <http://java.sun.com/j2se/javadoc/writingdoccomments>. The section about coding conventions is based on [Sun, 1997].

References

- [Bloch, 2008] Joshua Bloch. *Effective Java*. Addison-Wesley, 2008.
- [Lewis and Loftus, 2009] John Lewis and William Loftus. *Java Software Solutions* Foundations of Program Design. Pearson International, 2009.
- [Sun, 1997] Sun. Java code conventions, 1997. This document is freely available from <http://java.sun.com/docs/codeconv>.

5 For Friday

Study the notes, study Pages 80–87 of the book, and carry out the exercises on Pages 92 and 93 of the Book.