

BERUFSAKADEMIE MANNHEIM
Information Technology International

Lecture Notes

Programming I
Part 8 Styles and Conventions

Dipl.-Betriebswirt (BA) VOLKAN YAVUZ
{xmachina GmbH

November 2001–February 2002
Classes: TIM01AGR && TIT01EGR
Term: 1

Contents

1	Introduction	1
2	Programming Style	2
2.1	Formatting Conventions	2
2.2	Naming Conventions and Identifiers	3
3	Commenting	5
4	Documenting using Javadoc	6
4.1	Javadoc Tags	6
4.2	Javadoc Examples	7
4.2.1	Class Documentation	7
4.2.2	Field Documentation	7
4.2.3	Constructor Documentation	8
4.2.4	Method Documentation	8
	References	9

1 Introduction

It is very seldom that programs are never touched again after implementation, testing and being put into usage. In fact, programs have to be steadily corrected, adjusted to changing requirements, newly discovered bugs fixed, and so on. Thus, the program code has to be read (and understood) many times, not only by the original author, but potentially by many other programmers, too.

The readability of programs is influenced both by the programming language used and the *programming style* of the author. Programming style is – as the name suggests – to some extent a matter of personal taste.

The syntax of a programming language tells you what code it is possible to write – what the machine will understand. Style tells you what you ought to write – what the humans reading the code will understand. Code written with a consistent, simple style will be maintainable, robust, and contain fewer bugs. Code written with no regard to style will contain more bugs. It may simply be thrown away and rewritten rather than maintained. [Vermeulen et al., 2000, pg. xiii]

2 Programming Style

Principle of Least Astonishment Avoid things that will surprise a user of your software:

- Simplicity
- Clarity
- Completeness
- Consistency
- Robustness
- Document any deviations

Operators and Precedence Add parentheses whenever they contribute to clarity, even if they are redundant.

Leave optimization for last Do not waste your time optimizing code until you are sure you need to do it. The first step should always be on a clear and correct implementation. Given a clear implementation, optimization is – but only when necessary – performed easier than for code that has already been “optimized”. The “First Rule of Optimization goes: Don’t do it.” And remember the 80–20 rule: 20 percent of the code in a system uses 80 percent of the resources. If you are going to optimize, make sure it falls within the 20 percent portion.

2.1 Formatting Conventions

Indentation Indent nested code, so that the logical block structure becomes apparent:

```
1  class MyClass {
2      static {
3          ...;
4      }
5
6      void doSomething(int arg)
7      {
8          if (j < 0) {
9              ...;
10         } else if (j > 0) {
11             ...;
12         } else {
13             ...;
14         }
15
16         for (int i = 0; i <= j; i++) {
17             ...;
18         }
19
20         while (++k <= j) {
21             ...;
22         }
```

```
23
24     do {
25         ...;
26     } while (++k <= j);
27
28     try {
29         ...;
30     } catch (Exception e) {
31         ...;
32     } finally {
33         ...;
34     }
35
36     switch (value) {
37     case 0:
38         ...;
39         break;
40     default:
41         ...;
42         break;
43     }
44
45     // Anonymous inner class
46     button.addActionListener(new ActionListener() {
47         public void actionPerformed() {
48             ...;
49         }
50     });
51 }
52
53 class InnerClass
54 {
55     ...;
56 }
57 }
```

2.2 Naming Conventions and Identifiers

Use meaningful names Use names that have a meaning in the application domain you are coding for. In general, this means that the vocabulary of the problem domain – not that of the solution domain – should be used. For example, ‘CustomerList’ or ‘Customers’ is preferred over ‘CustomerArray’.

Type Names Identifiers for classes and interfaces start with an uppercase letter, and each word is capitalized..

- As *classes* represent *things*, use *nouns* as class identifiers.

```
58     class CustomerAccount ...
59     class KeyAdapter
```

- As collections or classes that group related things involve *many* things, pluralize these identifiers.

```
60     class CustomerAccounts ...
61     class Elements ...
62     class LineMetrics ...
63     class Beans ...
64     class Colors ...
```

- as *interfaces* declare the services provided or describe the capabilities of an object, use nouns or adjective for there identifiers:

```
65     interface ActionListener ...
66     interface Runnable ...
67     interface Accessible ...
```

Method Names Identifiers for methods start with a lowercase letter, and each word is capitalized.

- As *methods* perform *actions*, use verbs.

```
68     public void flush() ...
69     public Image getScaledInstance() ...
70     public void withdraw(int amount) ...
71     public void deposit(int amount) ...
```

- Follow the *JavaBeans* conventions for naming property accessor methods:

```
72     boolean isValid() ...
73     String getName() ...
74     String getAlias(int index) ...
75     void setValid(boolean isValid) ...
76     void setName(String name) ...
77     void setAlias(int index, String alias) ...
```

Variable Names Identifiers for variables start with a lowercase letter, and each word is capitalized.

- Use nouns to name variables.

```
78     class Customer {
79         ...
80         private Address billingAddress;
81         private Address shippingAddress;
82         private Phone daytimePhone;
83         private Vector openOrders;
84         ...
85     }
```

- Pluralize the names of collection references

```
86     Customer[] customers;
87     void addCustomer(int index, Customer customer) ...
88     Vector orderItems;
89     void addOrderItem(OrderItem orderItem) ...
```

- Establish and use a set of standard names for trivial “throwaway” variables.

Character	‘c, d, ’
Coordinate	‘x, y, z’
Exception	‘e’
Graphics	‘g’
Object	‘o’
Stream	‘in, out, InOut’
String	‘s’

- To distinguish between local variables and fields, always qualify field names with **‘this’**.
- When a constructor or **‘set’** method assigns values to a field, give the parameter the same name as the field, and qualify the field name with **‘this’**.

```
90     public Element(int key, String value)
91     {
92         this.key = key;
93         this.value = value;
94     }
```

Constant Names Identifiers for constant names are written in uppercase only, with words separated by underscore:

```
95     class Byte {
96         public static final byte MAX_VALUE = 255;
97         public static final byte MIN_VALUE = 0;
98         ...
99     }
```

3 Commenting

Commenting programs and code has the following intentions:

1. Write the documentation for two audiences:
 - For those who will use your code, so they can use it correctly and effectively.
 - For those who will maintain your code, so they can understand the implementation and safely modify and enhance it.
 - For yourself, so that you are able to understand why you did it that way.
2. Do not describe *what* the code does, but instead, describe *why* it is doing what it does. Do not repeat the code. The following comment only states the obvious.

```
100 // Apply a 5 percent discount to all invoices over a thousand
101 // dollars.
102 if (this.invoiceTotal > 1000.0) {
103     this.invoiceTotal = this.invoiceTotal * 0.95;
104 }
```

but does not answer the following questions:

- Why is the discount 5%?
- Who determined the discount and the dollar amount?
- When or why would these amounts change, and what else would have to be adjusted?

However, any domain-specific know-how should be explained:

```
105 // This term corrects for the effects of Jupiter, Venus, and
106 // the flattening of the earth:
107 sigma += (c1 * Angle.sin(a1)
108           + c2 * Angle.sin(Angle.minus(l1, f))
109           + c3 * Angle.sin(a2));
```

3. Insert comments *during* coding, and not after coding has been completed.
4. Update the comment whenever the code is updated. Wrong comments are even worse than no comments at all.

4 Documenting using Javadoc

The `javadoc` tool can be used to generate documentation in HTML (Hypertext Markup Language) format of classes and methods given a set of packages or individual `.java` source code files. As `javadoc` can also work on stubs of source files, i.e. work with empty methods, it can already be used in a very early stage of development to produce the API (Application Programming Interface) documentation.

`javadoc` comments enclosed within `/**` and `*/` and are placed directly before the declarative element to be documented. The following declarative elements can be documented:

- constructors and methods,
- fields
- classes and interfaces
- packages, and additionally, an
- overview over the whole set of packages

4.1 Javadoc Tags

Following are the generally available tags to be used in `javadoc` comments, and table 1 on the following page shows where each tag can be used. More information and stylistic guidelines can be found at Wri [2000], Req [1999].

<code>@author</code> $\langle name-text \rangle$	Adds an “Author” entry
<code>@deprecated</code> $\langle deprecated-text \rangle$	Adds a comment indicating that this API is deprecated, explained by $\langle deprecated-text \rangle$. A <code>@link</code> should point to the replacement API.
<code>@exception</code> $\langle class-name \rangle$ $\langle description \rangle$	<code>@exception</code> is a synonym for <code>@throws</code> . See the description given for <code>@throws</code> .
<code>@link</code> $\langle name \rangle$ $\langle label \rangle$	Inserts a link that points to $\langle name \rangle$, labelled by $\langle label \rangle$ just as the <code>@see</code> tag, but produces an in-line link rather than placing the link in the “See Also” section.
<code>@param</code> $\langle parameter-name \rangle$ $\langle description \rangle$	Adds a $\langle description \rangle$ for $\langle parameter-name \rangle$ in the “Parameters” section.
<code>@return</code> $\langle description \rangle$	Adds a “Returns” section with $\langle description \rangle$.
<code>@see</code> $\langle reference \rangle$	Adds a “See Also” heading with a link or text entry that points to $\langle reference \rangle$. Three forms are possible:
<code>@see</code> $\langle string \rangle$	Adds an entry for $\langle string \rangle$, but generates no link. This is suitable for resources not available online, like books.
<code>@see</code> $\langle a href=“\langle URL \rangle\#\langle value \rangle” \rangle$ $\langle label \rangle \langle /a \rangle$	Adds an entry for $\langle label \rangle$ and generates a link to the given $\langle URL \rangle$, which can be absolute or relative.
<code>@see</code> $\langle package \rangle . \langle class \rangle \#\langle member \rangle$ $\langle label \rangle$	Adds an entry for $\langle label \rangle$ and generates a link to the given $\langle package \rangle$, $\langle class \rangle$ or $\langle member \rangle$, or any valid combination thereof.
<code>@serial</code> $\langle field-description \rangle$	Describes a serializable field.
<code>@serialData</code> $\langle data-description \rangle$	Documents the sequences and types of data.

@serialField <i><field-name></i> <i><field-type></i> <i><field-description></i>	Documents the given <i><field-name></i> with <i><field-description></i> . There should be one @serialField tag for each component.
@since <i><since-text></i>	Adds a “Since” heading with the specified <i><since-text></i> .
@throws <i><class-name></i> <i><description></i>	Adds a “Throws” subheading to the generated documentation, and describes <i><class-name></i> .
@version <i><version-text></i>	Adds a “Version” subheading.

Table 1: Javadoc Tags and where they can be used

The fields are shown in “canonical” order as suggested by Wri [2000].

Tag	Overview	Package	Class and Interface	Field	Constructor and Method
@author	—	—	×	—	—
@version	—	—	×	—	—
@param	—	—	—	—	×
@return	—	—	—	—	×
@exception	—	—	—	—	×
@throws	—	—	—	—	×
@see	×	×	×	×	×
@since	×	×	×	×	×
@serial	—	—	—	×	—
@serialData	—	—	—	—	×
@serialField	—	—	—	×	—
@deprecated	—	×	×	×	×
{@link}	×	×	×	×	×

4.2 Javadoc Examples

4.2.1 Class Documentation

```

110 /**
111  * 1) Executive Summary <br>
112  * 2) State Information <br>
113  * 3) OS/Hardware Dependencies <br>
114  * 4) Allowed Implementation Variances <br>
115  * 5) Security Constraints <br>
116  * 6) Serialized Form <br>
117  * 7) References to any External Specifications <br>
118  *
119  * @author Author of this Class
120  * @version 1.0
121  *
122  * @see #getAvgCount
123  * @since 1.0
124  * @deprecated reason for deprecation
125  */
126 class Plain
127 {

```

4.2.2 Field Documentation

```

128     /**
129      * 1) What this field models.<br>
130      * 2) Range of valid values.<br>
131      * 3) Null Value.<br>
132      *
133      * @see #getAvgCount
134      * @since 1.0
135      * @deprecated reason for deprecation
136     */
137     public int x;
    
```

4.2.3 Constructor Documentation

```

139     /**
140      * 1) Expected behaviour<br>
141      * 2) State Transitions<br>
142      * 3) Range of Valid Argument Values<br>
143      * 4) Null Argument Values<br>
144      * 5) Range of return values<br>
145      * 6) Algorithms Defined<br>
146      * 7) OS/Hardware Dependencies<br>
147      * 8) Allowd Implementation Variances<br>
148      * 9) Cause of Exceptions<br>
149      * 10) Security Constraints<br>
150      *
151      * @param count Description of <code>count</code>
152      * @throws Exception Reason for <code>Exception</code>
153      * @see #x
154      * @since 1.0
155      * @deprecated reason for deprecation
156     */
157     public Plain(int count) throws Exception
158     {
159     }
    
```

4.2.4 Method Documentation

```

160     /**
161      * 1) Expected behaviour<br>
162      * 2) State Transitions<br>
163      * 3) Range of Valid Argument Values<br>
164      * 4) Null Argument Values<br>
165      * 5) Range of return values<br>
166      * 6) Algorithms Defined<br>
167      * 7) OS/Hardware Dependencies<br>
168      * 8) Allowd Implementation Variances<br>
169      * 9) Cause of Exceptions<br>
170      * 10) Security Constraints<br>
171      *
172      * @param count Description of <code>count</code>
173      * @return Description of return value
174      * @throws Exception Reason for <code>Exception</code>
175      * @see #x
176      * @since 1.0
177      * @deprecated reason for deprecation
178     */
179     public int getAvgCount(int count) throws Exception
180     {
181     }
    
```

List of Tables

1 Javadoc Tags and where they can be used 7

List of Figures

List of Acronyms

API Application Programming Interface

HTML Hypertext Markup Language

UML Unified Modelling Language

Very good formal guidelines on writing programs in the Java language are provided by Vermeulen et al. [2000]. Conceptual guidance on design and idioms are given in Warren and Bishop [1999]. A very concise checklist-style overview of UML (Unified Modelling Language) is given in Balzert [2001].

References

Requirements for writing java api specifications, 1999. URL <http://java.sun.com/j2se/javadoc/writingapispecs/>.

How to write doc comments for the javadoc tool, 2000. URL <http://java.sun.com/j2se/javadoc/writingdoccomments/>.

Heide Balzert. *UML kompakt – mit Checklisten*. Spektrum Akademischer Verlag, Heidelberg, Berlin, 2001. ISBN 3-8274-1054-1.

Allan Vermeulen, Scott W. Ambler, Greg Bumgardner, Eldon Metz, Trevor Misfeldt, Jim Shur, and Patrick Thompson. *The Elements of Java Style*. SIGS Reference Library. Cambridge University Press, Cambridge, 2000. ISBN 0-521-77768-2.

Nigel Warren and Phil Bishop. *Java in Practice – Design Styles and Idioms for Effective Java*. Addison-Wesley Publishing Company, Inc., Reading, Massachusetts, 1999. ISBN 0-201-36065-9.