Technical Note

Java Coding Convention

Brendon J. Wilson

April 4, 2000
Version 1.2

Copyright Notice

Table of Contents

Overview

Origin of this Document

This document is designed to provide Java developers with a generic set of coding conventions. By itself, a code convention does nothing; however, when combined with code reviews, proper object-oriented analysis and design, and eXtreme Programming practices, a consistent and thorough coding convention can help improve communication, reduce learning times for new team members, and encourage code reuse.

Intended Audience

This document is primarily intended for Java developers, but may also be of interest to development team managers, systems architects, and technical writers.

Relation to Other Systems

This document is based on the coding style that is prevalent in most JavaSoft example code, and should already be familiar to most Java developers. This document also outlines how to adopt these coding conventions by taking advantage of some of the capabilities of the VisualAge for Java integrated development environment from IBM.

Detailed Description

Coding Style

Coding style refers to the method of formatting code in a manner that promotes organization, maintainability, and readability. The elements of this formatting in Java include the method of indentation, declaration of import statements, and declarations of variables.

Tabbing and Indentation

The West Coast tabbing convention is the preferred style for Java development, as opposed to C-style (K&R) tabbing convention. The tab indent should be 4 spaces, although it is preferred that tabs are used rather than spaces for the actual indentation. A correctly indented piece of code using the West Coast style:

```
try
{
    size = inStream.available();
}
catch (IOException e)
{
    throw e;
}
```

Note that opening braces always begin on a new line, and that compound statements (such as catch) also begin on a new line. VisualAge for Java (VAJ) can be configured to automatically format your code as you type; to configure VAJ:

1. Start VisualAge for Java.
2. Select **Window | Options**.
3. Select **Coding**, and ensure the tab spacing is set to 4 spaces.
4. Select **Coding | Formatter**, and check both checkboxes.
5. Select **Coding | Indentation** and select **Smart Auto** Indent.
6. Click **OK**.

Import Statements

Despite requiring additional coding effort, the preferred style for import statements is to provide fully qualified class names, as opposed to wildcard imports. For example, instead of:

```
import java.io.*;
import javax.mail.*;
import com.brendonwilson.*;
import java.awt.*;
import java.applet.*;

the import statements should be specified completely:

import java.applet.Applet;

import java.awt.Button;
import java.awt.Label;
import java.awt.Panel;

import java.io.FileInputStream;
import java.io.IOException;

import com.brendonwilson.FooClass;
```

Note that each class definition required by the class is explicitly imported, and that imports are grouped by package; note also that import statements for standard Java libraries, and Java standard extension libraries come first in the import block. The groups are alphabetically sorted, as are the individual imports statements within a particular group, and each group of package import statements is separated from the following group by a single empty line.

The import statement block should appear at the very beginning of the Java source file, before the class prologue documentation. The final import statement should be separated from the class prologue by two empty lines.

```
import java.io.FileInputStream;
import java.io.IOException;

import com.brendonwilson.foo;


/**
 * Class prologue documentation.
 */
```

The reason for creating import statements in this manner is to reduce the possibility of error due to a namespace clash, as well as make the dependencies of code on specific libraries explicit. Should specific core libraries in Java change, it’s easy to determine which classes in our code are affected.

Variable Declarations

In addition to giving variables meaningful names, thought should be given to where variables are declared. As a general rule, local variables should be declared at the beginning of the block of code in the following manner:

```
public String format(Message message,  oolean encrypt)
{
    StringBuffer out = new StringBuffer();

    if (encrypt)
    {
        message.setBody(encrypt(message.getBody());
        return format(message, false);
    }
    else
    {
        String to = “To:” + message.getToField();
        String from = “From:” + message.getFromField();
        String to = “Cc:” + message.getCcField();

        out.append(to);
        out.append(from);
        out.append(cc);
```

```
            out.append(“\r\n”);
            out.append(message.getBody());

            return out.toString();
        }
    }
```

Notice that the variables are declared at the beginning of the block in which they are required, which is either immediately after the declaration of the method, or the if/else statement. The exceptions to this rule are the declaration of Exceptions in a catch statement, and the declaration of index variables in for loops:

```
    try
    {
        for (int I = 0; I < addresses.length; I++);
        {
            System.out.println(addresses[I]);
        }
    }
    catch (Exception e)
    {
        throw e;
    }
```

It is acceptable for the index of the loop to be declared in the loop; however, if the index variable is defined before the loop, it should be declared at the beginning of the block enclosing the loop.

Naming Conventions

Class and Interface Names

Class and Interface names begin with a capital letter, with subsequent words within the name also capitalized. Class and Interface names should provide some clue as to their purpose; good examples:

```
    ThreadManager
    MouseListener
    Cipher
```

Bad examples of class names include:

```
    myclass
    mainApplication
    DOIT
```

Although all naming should avoid using abbreviations in order to preserve readability, well known acronyms may be used in class names. In this case, the acronym letters should be capitalized, as should first letter of any following word in the class name. For example, it would be acceptable to have a class name of HTTPRequest, but HttpRequest, Httprequest, or HTTPrequest would not be acceptable.

Method Names

Unlike class names, methods should begin with a lowercase letter, with following words properly capitalized. The method name should typically consist of an action verb, followed by the name of an object or property; ideally, it should be possible to read a call to the method aloud in plain English with little or no modification. For example:

```
rotateAround(Axis a, double angle);
```

is a good name; a method call can be read as "rotate around axis a by angle degrees". While:

```
rotate(double angle, Axis a);
```

obscures the purpose of the method; it's not clear what's being rotated (the object itself, or one of its parameters) make interpretation of the call difficult. In keeping with the Java Beans code convention, common properties should be accessed using get and set methods. For example, an object with a property called timeout might have a set method with the following signature:

```
public void setTimeout(long seconds);
```

It's clear from this method name that a call will set the timeout property on the object to be the given number of seconds.

Variable Names

In a similar fashion to methods, variable names should begin with a lowercase letter, with proper capitalization on subsequent words:

```
int timeout;
Property propertyTable;
Application mainApplication;
```

Constants are the exception to this rule, and are detailed in the next section.

Constants

In order to differentiate class or instance constant variables from regular instance variables, constant variables use all uppercase, and individual words are separated using an underscore:

```
static final int HTTP_OK_RESPONSE = 200;
static final String GNUTELLA_CONNECT = "GNUTELLA CONNECT\n\n";
```

Documentation

Besides providing external documentation, developers are required to adequately document all code they produce. This documentation includes not only inline documentation, but also variable, method, and class level documentation.

Class Prologues

In order to facilitate the documentation of a class's purpose, each class should include a brief piece of documentation just after the import statements. The class prologue should indicate dependencies on other classes, special notes, and any assumptions made by the class designer.

In addition, the author, version, creation date, and copyright notice should be included using the standard JavaDoc tags. Any additional notes, dependencies, or 'see-also' notes should also be included using the appropriate JavaDoc tags.

A typical class prologue would look like:

```
/**
 * The first line is a summary of the class's purpose.<p>
 *
 * The second paragraph details specifics of the class's
 * operation, and any special dependencies, or protocols
 * developers should know about when using the class.<p>
 *
 * @author      Author's Name
 * @date        January 1st, 2000
 * @version     version number
 */
public class Foo extends Bar
```

Notice that HTML tags are acceptable in documentation, as all documentation will be compiled using the JavaDoc tool, which produces HTML API documentation. You can configure VAJ to automatically provide skeleton documentation when you create a new variable, method, or class; to configure the default documentation skeleton VAJ adds to each class created using the toolbar 'Create Class' button:

1. Start VisualAge for Java.
2. Select **Window | Options**.
3. Select **Coding | Macros**.
4. Edit the skeleton class prologue.
5. Click **OK**.

Method Prologues

Similar to class prologues, method prologues should appear before each method declaration, describing the purpose of a method. The method prologue should include JavaDoc tags to information on the method arguments, return value, and if applicable, possible exceptions.

A typical method prologue would look like:

```
/**
 * Description of the method's purpose, and any special
 * circumstances which apply to when the method should or
 * shouldn't be called.
 *
 * @param       param1 description of parameter 1.
 * @return      description of the return value.
 * @exception   ExceptionClass description of exception.
 */
public boolean doIt(int param1) throws ExceptionClass
```

As with class prologues, VAJ is capable of generating skeleton method prologues to insert at the beginning of methods created using the 'Create Method' button. To configure the default documentation skeleton VAJ adds to each class created using the toolbar 'Create Class' button:

1. Start VisualAge for Java.
2. Select **Window | Options**.

3. Select **Coding | Macros**.
4. Edit the skeleton method prologue.
5. Click **OK**.

Variable Documentation

Variable documentation is only really necessary on class or instance variables, and should follow the JavaDoc convention:

```
/**
 * Description of the variable’s purpose.
 */
public String myString;
```

For the most part, the purpose of local variables should be documented as a part of the inline method comments; a local variable should only be commented if its purpose is not immediately apparent.

Inline Code Comments

Inline code comments should be included to guide the unfamiliar developer with the operations being performed within a method. With the exception of comments that block out old and unused sections of code, all inline code comments should use the double-slash format of comments:

```
// This is a comment.
```

As opposed to the slash-asterisk format of comments:

```
/* This is a bad comment. */
```

The reason for using the former format is simple: if a large section of code needs to be commented out at a later time, the latter format will result in errors, as the compiler attempts to reconcile the open and closing of comment sections. Inline code comments should always appear on a line by themselves.

In addition to providing adequate inline comments to guide developers, special inline comments should be used to alert developers to areas of code that are based on questionable practices, or need to be redone at a later date. To comment areas which contain code that needs to be revisited at a later date for implementation:

```
// TODO: Description of what needs to be done.
```

To comment areas of code that need to fixed to operate in a more desirable fashion at a later date:

```
// FIXME: Description of required fix.
```

Sections of code that may not yet be complete due to an unanswered question, or an unproven assumption should be commented:

```
// RESOLVE: Description of the question yet to be answered.
```

Best Practices

Use the StringBuffer Class

To often in Java code, developers overuse the '+' operator in order to concatenate Strings; unfortunately, this practices inadvertently affects performance. For example, when creating a long string, a developer may be inclined to write:

```
String foo = “This”;
foo = foo + “ is”;
foo = foo + “ bad!”;
```

In this example, the Java compiler transforms the '+' operator into a series of StringBuffer calls:

```
String foo = “This”;
foo = (new StringBuffer(foo)).append(“ is”).toString();
foo = (new StringBuffer(foo)).append(“ bad!”).toString();
```

As is obvious from the example, each use of the '+' operator results in the creation of a StringBuffer object, a call to append(), and another call to toString(). It is more efficient to construct the string using a StringBuffer, use the append() method, and perform the conversion to String as the very last step in the process:

```
StringBuffer foo = new StringBuffer(“This”);
foo.append(“ is”);
foo.append(“ good!”);
```

In addition, as append() returns the result of the concatenation, each of these calls can be combined together:

```
StringBuffer foo = new StringBuffer(“This”);
foo.append(“ is”).append(“ good!”);
```

Avoid Catching Exception

In attempting to write error-free code, some developers may be tempted to catch Exception without propagating a more specific Exception subclass up to higher levels of the program. This practice should be avoided for several reasons:

1. It makes it difficult to debug software if Exceptions are never propagated.
2. It makes it masks the expected types of exceptions that may occur during normal application operation.

Specific exceptions should be caught and the exception either rethrown, translated into another exception and thrown, or handled appropriately. In some applications, the number of extraneous exceptions which may be thrown in a section of code might make it overwhelming to have a catch block for each exception type; in this situation, it is acceptable to catch Exception only if the catch block includes documentation describing the expected Exception subclasses that the block has been designed to replace.

GNU Free Documentation License

In order to limit the size of this document, a copy of the GNU Free Documentation Version 1.1 has not been included within the document itself. A copy of the license is freely available for download from the Free Software Foundation:

http://www.gnu.org/copyleft/fdl.html

Alternatively, a copy of the license terms can be obtained by writing the Free Software Foundation at:

Free Software Foundation, Inc.
59 Temple Place, Suite 330,
Boston, MA
02111-1307
USA

References

IBM Corporation. VisualAge Developer Domain [online]. Rochester, NY: International Business Machines, 2001 [cited 3 May 2001]. Available from the World Wide Web: (www.ibm.com/vadd)

Revision History

Version	Author	Date	Description
1.0	Brendon J. Wilson	April 4, 2000	Document Created
1.1	Brendon J. Wilson	May 7, 2001	Added section on class, method, variable and constant names.
1.2	Brendon J. Wilson	June 9, 2001	Added GNU license info.