Coding Standards for Java

New England Java Users Group
http://www.nejug.org

March, 2002

2nd Revision

Preface

We have tried to provide a usable guide to establishing and following coding standards.

This guide is intended for any Java developer seeking guidance in establishing standards in their own work, for their project team, and in their entire organization.

Our goal is to facilitate universal acceptance of our recommendations within your project team, department, or organization. To that end we have tried to produce a small, approachable document. To enhance its use as a reference document we have created an extensive index. We have also organized the individual points so that you can quickly agree on the basics and focus your discussions on substantive issues only.

We assume you care about doing a good job and achieving excellence in your work. We understand that following these guidelines will not guarantee your code will be excellent. However, we believe that ignoring them will make it much harder to attain excellence.

You may resist the restrictions implied by these guidelines and they may seem at first overly structured. Nevertheless, if your team adopts them we hope you will find that this structure enables you to shine and empowers you to excel. Rather than constrain us, these guidelines provide a safer playground in which we can create- without worrying about details which can drain our energy.

The New England Java Users Group

The New England Java Users Group was formed in November 1998 by Mark Richards and Bill Rushmore of Apex Consulting Group, Inc. with the purpose of providing a forum for exchanging ideas and discussing various topics and issues relating to Java technology. The group is open to all individuals, and membership, meetings, and refreshments are free of charge.

With over 1800 registered members as of January 2002, the NEJUG is one of the largest Java Users Groups in the country. Meetings are held once a month, with the exception of July and December. The speakers at each meeting include Sun engineers, Java technical authors, technical resources from local vendors, and NEJUG members. Past speakers have included Peter Coad, Martin Fowler, Peter Haggar, and Ken Arnold. The meeting topics are technical in nature, and involve some aspect of Java and J2EE. The NEJUG web site is located at www.nejug.org. It provides a complete list of all past, present, and future meeting topics, as well as presentation slides from each meeting. Also at the web site you can find general information about the group, meeting details, directions, meeting registration, membership registration, a book and product review section, and a NEJUG Bulletin Board.

The administrative body of the group consists of a president and two officers. Mark Richards, Chief Architect at Apex Consulting Group, Inc. is the NEJUG President (nejug@apexcgi.com). The NEJUG Officers are Robert "Red" Rogers, System Engineer from Sun Microsystems (red@nejug.org), and Donna Alger, a Web Architect and Manager of the Maine Java Users Group (donna@nejug.org). Mark Richards is the primary NEJUG contact, and can be reached by e-mail at nejug@apexcgi.com.

The sponsors of the New England Java Users Group include Apex Consulting Group, the founder and primary sponsor, Sun Microsystems, Sun Education, Addison-Wesley, and JPMorgan. As the primary sponsor, Apex Consulting Group, Inc. (www.apexcgi.com) provides group leadership, web site administration and maintenance, marketing materials, meals and refreshments. Sun Microsystems (www.sun.com) provides the user group with a facility to meet once a month, as does JPMorgan. Sun Education also sponsors the group by providing meeting space for the NEJUG Special Interest Groups. Addison-Wesley (www.awl.com/aw) provides the group with technical Java books relating to each meeting topic, as well as pre-release books, Java Class Library posters, and other technical resources given away at each meeting.

For more information about the New England Java Users Group or to become a member, please visit the NEJUG web site at www.nejug.org.

Credits

The Java Coding Styles and Guidelines Special Interest Group was formed as part of the New England Java Users Group in January of 2001 with the purpose of discussing and investigating the various standards, styles, and guidelines that are used in Java. After several meetings, the group decided to create a Java standards, styles and guidelines document to share with the rest of the user group and Java community. After several more meetings, the SIG decided to formalize the document and publish it in book form. The book you are reading is a result of the hard work, efforts, and dedication of the Special Interest Group members.

The following are brief biographies of the special interest group members who dedicated their time, thoughts, and efforts to produce this guide:

Doug Chamberlin, Editor
dchamberlin@andoversoftware.com

Doug Chamberlin has contributed to the productivity of programming teams with insight and humor for over 25 years as a professional software developer and project lead. He is currently an Associate with http://www.dlawton.com and spends much of his time trying to keep up with current technology.

Mark Johnson
mfjohnson98@yahoo.com

Mark first started programming in the late 70's writing medical software for Apple II's. Currently enjoying the power and flexibility of Enterprise Java and object oriented programming while building strategic sourcing systems.

Theophano Mitsa
TheoMitsa@aol.com

Dr. Theophano Mitsa has 10 years experience in academia and industry in the areas of software development and image processing. She is the author of 38 technical publications and holds 7 US patents. She is currently a software consultant with Research Corporation Technologies.

Sean Murphy
virtualsean@fastdial.net

Chip Pate
cpate3@yahoo.com

Chip graduated from University of Southern Maine in 1999 receiving the Outstanding Student in Computer Science Award for the cumulative work throughout his years there. Since then he has worked as a consultant for wireless networks and for Intel. Currently he enjoys discovering new technologies that push the envelope of the way we do things today.

Mark Richards
wmrichards@worldnet.att.net or nejug@apexcgi.com

Mark Richards is the President of the New England Java Users Group, and also a Chief Architect at Apex Consulting Group, Inc. He has been involved in software design and development since 1984, and since 1996 has served as a lead developer and architect on Internet/Intranet and B2B projects using Java and J2EE.

Mark was the President of the Boston Java Users Group from 1997 to 1998, and is a Sun Certified Java Programmer, a Certified Java instructor, a BEA WebLogic Certified Developer, and has passed the first part of the Sun Certified J2EE Architect certification. Mark holds a Master's Degree in Computer Science from Boston Univerity.

Bill Rushmore
rushmore230@charter.net

Bill Rushmore is a long time Java developer and advocate. He is one of the original founders of the NEJUG and the first NEJUG president.

Brian Tarbox
btarbox@world.com

Brian Tarbox has been leading development teams for over twenty years with two patents and two InterOp Product of the Year awards. He is CTO of Been There - Done That Software, LLC.

Suzanne Trayhan
zanne@attbi.net

Venugopal Vasireddy
haripriya2@hotmail.com

Quan Yang
qqyang_2000@yahoo.com

Dr. Quan Yang has worked in telecommunication companies such as GTE, Nokia and in pharmaceutical companies such as Pharmacia as a software engineer and project lead. He is now with the Genome Therapeutics Corp.

Hong Zhuang
zhuang@world.std.com

Hong Zhuang is a J2EE consultant. She has helped companies such as State Street Bank, Thomson Financial, and Fidelity build distributed mission-critical enterprise applications.

Thanks to Dennis Kenny for editing assistance!

Apex Consulting Group is a Professional Services firm that delivers unique enterprise solutions from systems integration to digital business models across a broad spectrum of leading technologies for top companies in the New England region. Apex Consulting Group is the founder and primary sponsor of the New England Java Users Group, and the sponsor of this Java Standards and Guidelines book.

The Apex Consulting Group solution is business-oriented and technology-based. Apex Consulting Group works with its clients to identify business goals and designs the business processes and technical solutions to meet those goals and is focused on delivering measurable ROI, profitability and customer loyalty. Apex Consulting Group cost-effectively meets the ever evolving and increasingly demanding needs of our clients with the highest level of quality delivered by senior business and technology professionals.

In the area of Java and J2EE technology, Apex Consulting Group's goal is to help companies in the Greater Boston area develop and deploy robust, quality J2EE and Java-based applications by providing high quality technical and information management consulting services. Our focus is on developing and deploying J2EE-based architectures with an emphasis on reliability, performance, and scalability. To design these architectures we rely on JavaServer Pages, Servlets, Enterprise JavaBeans, Java Messaging, Java Transaction Processing, Security, and JDBC.

Apex Consulting Group also has expertise in the development of non-J2EE Java-based architectures, with a special emphasis on high-volume and high-speed transaction processing. Within this area we rely on experience designing and developing applications such as high-speed financial transaction processing systems, large-volume order processing and fulfillment systems, and high-volume automated hardware testing systems. In each of these areas Apex has met or exceeded client expectations.

To learn more about Apex Consulting Group, please visit their web site at http://www.apexcgi.com or contact Tom Stephanian at 781-944-0212.

Apex Consulting Group, Inc.
P.O. Box 636
Wilmington, MA 01887

Tel: 617-489-9000
Fax: 781-944-1988
email: info@apexcgi.com
web: www.apexcgi.com

Table of Contents

Overview

Organization

This document lists our recommendations for coding Java. Each recommendation is numbered, explained, and examples are provided.

We acknowledge that developers have strong feelings about many of these recommendations. These feelings are the single most significant factor in preventing recommendations from being adopted by groups of developers. Achieving a consensus is sometimes very difficult.

To help achieve consensus, we have classified the recommendations into either a *standard*, a *style* or a *convention*. We expect the *standards* to be easy to agree on. We expect the *styles* to be harder to agree on but our discussion of the choices should help. The relative freedom you have implementing *conventions* should make them easier to adopt.

Each group is presented in a separate chapter.

Standards

Standards are those recommendations which are thought to be so universal that they are strict requirements. One test for this classification is whether any Java developer would immediately expect the standard to be followed. Another test is how violations are dealt with. We expect that code which contains violations of standards will be rejected during a code review and not be allowed to reach production status.

Styles

Styles are recommendations for which there is some legitimate disagreement among experienced developers. For each style recommendation we present our views on the alternatives. We expect one of the alternatives will be chosen and adopted by you. One example of this is placement of braces (STY-12). We opted for one style to be used for all examples in this document.

Conventions

Conventions are recommendations in areas where you should make a selection so that your coding will be consistent and not haphazard. For example, we recommend you set a maximum number of lines that a method can contain. What that maximum value should be is up to you.

Principles

All decisions made regarding these recommendations should be made keeping certain principles in mind. These are the principles which have guided our discussions and which we feel are the most relevant.

Balance

Often the goals being sought in determining the best recommendations are in conflict. When this happens a balance must be struck which allows the essence of each goal to prevail. For example, some code may be originally well-structured but run slowly. In modifying the code to increase performance, the original structure sometimes must be altered. The developer must balance the need for increased performance with the need to retain an understandable structure.

Brevity

Succinct expression is appreciated. However, overly terse expressions are to be avoided. (This is another example of balance!)

Uniformity

Recommendations of similar types should be adopted as a group. Recognize patterns in the recommendations and promote them.

Consistency

Maximize the reader's understanding when they read your Java code and minimize surprises by applying these recommendations consistently.

Readability

Code should be written to be readable. Recommendations which are made mainly to enable coders to write more efficiently should be avoided. Code which is structured to enhance runtime performance over readability should also be avoided. Review for performance *after* ensuring that the code is well-structured, understandable, and works correctly.

References

We have made reference to several previous documents and articles. The most prominent of these is the *Code Conventions for the Java™ Programming Language* document from Sun which we refer to via [Conventions]. See the References list on page 5-1 for details on other references.

Standards

STD-1 Package naming.

Follow the nearly universal naming convention for packages described in [Conventions] section 9. Package names contain only lower case letters. Never create a package which uses a package prefix already being used by another entity, for example “javax”.

Proper package name examples:

```
package java.util;
package java.io;
package org.w3c.xml.parser;
package com.mycom.mypackage;
```

Improper package name examples:

```
 // Name should be lower case
package Com.MyCom.Mypackage;

// "javax" has already been used
// MyPackage should be all lower case
package javax.Mypackage;

// Name should be lower case
package com.myCom.Mypackage;
```

STD-2 Class and Interface naming.

Again, following the guidelines in [Conventions], class names should be nouns using mixed case with embedded words capitalized. Do not use embedded underscores within names.

Avoid acronyms and abbreviations unless they are *already* more widely used than their long form (e.g. HTML). When using acronyms do not capitalize the whole acronym. Instead treat it as a word and only capitalize the first letter.

Interface naming follows class naming exactly.

Proper class name examples:

```
MyDriver
MyClass
HtmlConverter
```

Improper class name examples:

```
myDriver    //Name should start with capital letter
My_Class    //Name should not have embedded underscores
myclass     //Name should start with capital letter
Transform   //Names should be nouns
```

STD-3 Method naming and formatting.

Methods are active and, therefore, should be named using verbs. As with class names, use mixed case except the initial letter is always lower case.

No spaces should exist between a method name and the opening parenthesis of a parameter list.

Proper method name examples:

```
getX()
createX(x)
updateMyTable()
```

Improper method name examples:

```
GetX()              // Name should start with
                    // lower case letter
create X(x)             // Name should not have
                    // space within the name
updateMyTable ()    // Space between name and
                    // the parenthesis
converter()         // Names should be verbs
```

STD-4 Variable naming.

Variable names should follow those of methods. Do not begin variable names with dollar signs ($) although the compiler will permit this.

Proper variable name examples:

```
myVariable
```

Improper variable name examples:

```
$my_variable    // No dollar sign should be at the beginning
MyVariable      // Name should start with lower case letter
my_Variable     // Name should not have embedded underscore
```

STD-5 Constant naming.

In order to make constants stand out in the code they are named with all capital letters. In order in increase the readability of the names, separate embedded words with a single underscore.

Proper constant name examples:

```
MY_CONSTANT_A
INCHES_TO_CENTIMETERS_FACTOR
```

Improper constant name examples:

```
Conversion_Factor_A     //Name should be all capital letters
MyMagicValue            //Name should be all capital letters
my_constant_b       //Name should be all capital letters
```

See CON-18 for more on constants.

STD-6 Use of JavaDoc comments is required.

Because the public interface to a class is key to being able to use the class effectively, JavaDoc comments are not optional for public classes and methods. The specific formatting and minimum content may be open for discussion but the presence of JavaDoc comments is not.

JavaDoc comments are important in that they provide documentation which is external to the source code and which can be easily maintained.

STD-7 Use of implementation comments is required.

JavaDoc comments are intended to define and describe the public interface of a class or method. Implementation comments are for describing the internal implementation.

All code should be commented to explain the implementation techniques used and the reasons *why* the code was written the way it was. Exactly how those comments appear is a stylistic question (see STY-19). How thoroughly the comments cover the material is a convention (see CON-11). However, the need for comments is beyond debate.

STD-8 Consistency of formatting is required within a source file.

Sometimes you acquire source code which differs from the accepted format you usually employ, including your choices for indentation, use of white space, etc. When this happens resist the impulse to mix your personal adopted format with the existing one. Respect the existing code and format your additions consistent with the existing format. Then pursue the option to reformat the entire source file as a separate task.

Mixed formatting styles are never acceptable because it confuses the reader and contributes significantly to misunderstanding. Remember, there are a number of styles described in this guide which require making a choice. Respect the choices of others.

STD-9 Avoid local declarations which obscure declarations at higher levels.

Do not declare a block variable with the same name as a method or class variable. Doing so unnecessarily obscures the code.

Styles

STY-1 Order sections within source files consistently.

Java source files should always have the following sections in the following order:

1. Package or file-level comments.
2. Package and import statements.
3. Public class and interface declarations.
4. Private class and interface declarations.

STY-2 Order of import statements.

Use the following order:

1. Standard packages such as java.awt, java.io, etc.
2. Third party packages such as com.ibm.xml.parser.
3. Your own packages.

Within each of the above groupings order the packages in alphabetic order.

STY-3 Import statement detail.

Two schools of thought exist for how to declare import statements. The first school says to use the * form to reduce the number of import statements. This also makes it much easier to introduce the use of classes located within the packages already imported because a new import statement does not need to be added.

The alternate view is that individual import statements for each imported class makes the origin of each class explicit and unambiguous.

We recommend use of * for standard packages, reserving explicit class imports for your own classes or those of third party vendors.

STY-4 Ordering of class parts.

Class declarations have the following sections in the following order:

1. Javadoc comments.
2. Class declaration statement.
3. Class-wide comments.
4. Class (static) variable declarations in the following order:
    1. Public

2. Protected
3. Package level
4. Private

5. Class instance variable declarations in the same order (public, protected, package level, private).
6. Method declarations. See the following style for the ordering of methods.

See [Conventions] 3.1.3

STY-5 Ordering of methods within classes.

Some like constructors to appear first, with other methods following in alphabetic order. Others like all the methods to appear in alphabetic order. A third choice is to group methods according to some measure of their functionality. However, the goal should always be to make the methods easy to locate.

It is arguable that constructors are the most important methods of a class and one is *always* used when the class is used. They will, therefore, always need to be located and referenced by a user of the class and should be prominently located in the source code.

For others, the consistency of a purely alphabetic ordering of all methods is more appealing. Locating constructors is never a problem because they are always located using the same procedure as locating any other method. Furthermore, JavaDoc documentation is the proper place to reference constructor details so the location in the source code is of less importance.

We recommend placing constructors at the top of a class and follow them by a simple alphabetic ordering of other methods or a function grouping of other methods. If you use a functional grouping, then document your placement logic somewhere easy to find.

STY-6 Limit length of source code lines.

You cannot assume a printed page or someone else's display window will be as wide as yours. However, you can assume a minimum 72-80 character display width.

Therefore, in order to ensure that your code displays properly you must limit the length of lines to a reasonable amount. Our strong suggestion is to use an 80 character working limit.

STY-7 Line continuation of method signatures.

When formatting a method signature, which is one of the code constructs which will naturally become quite long, break it before the method identifier and indent continuation lines twice the usual amount.

Example:

```
public static void preformAction(String arg1,
    int arg2, String arg3, Object arg4, String arg5)
{
  …
}
```

Note the trailing comma on the first line. The need for this is mentioned in the next item (STY-8).

STY-8 Line continuation of general code.

Break lines which extend beyond the line length limit at places which suggest a continuation exists. This helps the reader follow the entire statement. For example, a trailing comma at the end of a source line suggests that the statement is continued on the next line. Likewise, a trailing operator suggests another operand follows.

Example:

```
String createTableCoffees =      //ends on the next line
  "CREATE TABLE BOOKS " +
  "(BOOK_NAME VARCHAR(32)," +
  "BOOK_ID INTEGER, PRICE FLOAT, " +
  "SALES INTEGER, TOTAL INTEGER)";
```

If possible, break lines at higher levels of grouping rather than at lower levels.

See section 4.2 in [Conventions] for good examples.

STY-9 Indentation levels.

Indentation of code should be done using a uniform amount. Common indentation amounts of 2, 3, 4, or 8 spaces should be used. Pick one and stick to it.

STY-10 Indentation using tabs.

Hard tab characters should not be used to indent code. They force the reader to set tab stops to a value which conforms to your indentation level in order to see the same source indentation you are seeing. Using spaces ensures the source code remains formatted as it was intended.

[Conventions] in section 4 allows the use of tab characters for indentation but also specifies 4 spaces as the indentation amount and that tab stops *must* be set every 8 positions. These

specifications do not work well together, so we disagree with [Conventions]. Tabs should not be used to indent source code.

Some argue that the use of tabs reduces the size of source files but this argument has little merit relative to the value of ensuring the code is always properly formatted.

STY-11 Indentation of controlled statements.

Compound statements (if, while, etc) include a controlled statement block which should always be indented an indentation level.

STY-12 Brace placement.

Placement of braces relates to proper indentation. Two major positions exist on the placement of braces. The first proposes placing of the opening brace at the end of a line of code. The second proposes placement at the beginning of the following line.

Arguments supporting the end-of-line style are:

1. Reduces the number of lines of code, allowing more code to be seen. Most books and magazines use this style simply to conserve space.
2. The trailing brace suggests to the reader that the statement controls code appearing below. It is a visual manifestation of the logical structure which is then reinforced by indenting the controlled code.
3. Some fonts render the opening brace character so thinly it can be missed on the page, making the line appear to be all white space.
4. Sun supports this style.

Arguments supporting the next-line style:

1. Makes visual locating of the matching brace easier because they are always positioned at the same indentation level.
2. Many programmers coming to Java from C++ are used to this style.
3. Since if/for/while statements are often long, finding the opening brace on the next line can be easier than at the end of the line.
4. Makes the location of braces more predictable. The opening brace is always in the same place relative to the control structure rather than floating at the end of a line.

These positions are difficult to resolve so we make no specific recommendation. Pick one and stick to it.

STY-13 Ternary statement usage.

Ternary statements can easily become difficult to read. Limit their use to single line, simple cases only. Never nest ternary statements.

STY-14 Always use a break statement in each case.

Be sure to include a break statement for each case in a switch statement. Although it is optional, omitting the break statement is an error-prone technique.

STY-15 Include a default case in all switch statements.

This is a basic defensive programming technique. The default case exists in the Java language for a reason. Include the default case so the unexpected is handled, but log the fact that it happened so you are aware that the 'impossible' case really did occur. At the very least throw an exception to alert the application that an unexpected condition occurred.

Example:

```
int thisColor = getStoneColor();
switch(thisColor)
{
  case 0 :
    stones[index] = Color.green;
    break;
  case 1 :
    stones[index] = Color.red;
    break;
  case 2 :
    stones[index] = Color.blue;
    break;
  case 3 :
    stones[index] = Color.yellow;
    break;
  default:
    stones[index] = Color.white;
    System.out.println("got unexpected stone color, defaulting to white");
    break;
}//switch
```

STY-16 Initialize local variables where they are declared, but only for non-default values.

The only reason not to initialize at the point of declaration is when a computation is required which cannot be performed at that point.

Explicitly re-initializing variables to default values is redundant, inefficient, and unnecessary. Know the default initializations and learn to depend on them. If you want to highlight the fact that a specific value is assumed then add a comment.

Example:

Here the initialization is a problem because one constructor initialized the m_thread variable while depending on the speed variable to be 0 and the other does *not* initialize m_thread but overrides the initialization of speed. Finally, the JVM already initializes the speed variable to 0 so explicitly doing so in the code makes the code larger and adds to the execution time every time this class is instantiated.

```
Class Foo
{
  private Thread  m_thread ;
  private int speed = 0;

  Foo()
  {
    m_thread = new Thread();
  }

  Foo(int x)
  {
    speed = x;
  }
}
```

Better coding would make the initialization more uniform:

```
Class Foo

{
  private Thread  m_thread = new Thread();
  private int speed;

  Foo()
  {
  }


  Foo(int x)
  {
    speed = x;
  }
}
```

Here the existence of the second constructor is clear. It is there so the speed value can be set to a specific value as soon as the Foo object is created.

See Praxis 37 in [Haggar00] for a good treatment of initialization issues.

STY-17 Initialize members and sub-objects either in a declaration or in constructors.

Classes which perform some initialization in declarations and some in constructors can be difficult to debug. If a variable is given a value in a one of several constructors then it should be initialized in a similar manner in all constructors. In this case the variable need not have an initialization in its declaration. Not only is it confusing to perform an initialization which is almost immediately replaced with another but it is redundant and therefore inefficient.

STY-18 When commenting out code, only use // style comments

One useful practice is to temporarily comment out blocks of code. However, this can be difficult if the code contains a mix of comment forms because the existing embedded comments can interfere with the outer comment marks. This is particularly problematic using /* .. */.

Furthermore, using /* .. */ to comment out large blocks leaves the interior code of the blocks themselves largely unchanged and often looking like it is still active code. While the widespread use of IDE environments with color-coded syntax editors helps with this problem, it is still an error-prone activity.

Finally, the widespread use of change management systems (a.k.a. version control systems) makes this practice somewhat obsolete since previous versions of files which contain removed code can always be recovered.

We recommend change management systems be used, even for single developer projects, and therefore do not recommend commenting out large blocks of code.

However, if you must continue this practice, consider the manner in which you perform this step. The one comment form which will always work for commenting out code is the trailing comment form (i.e. using //). Therefore, you should always use this form to comment out code by prefixing each line with the // at the very beginning of the line.

One good side effect of using the trailing comment form is that every line which is commented out is clearly marked as such. This help prevent the reader from inadvertently thinking commented out code is active.

Example:

```
//    /**
//     * Converts this <code>Date</code> object to a
//     * <code>String</code> of the form:
//     * <blockquote><pre>
//     * dow mon dd hh:mm:ss zzz yyyy</pre></blockquote>
//     *
//     * @return  a string representation of this date.
//     * @see     java.util.Date#toLocaleString()
//     * @see     java.util.Date#toGMTString()
//     */
//    public String toString() {
```

```
//   DateFormat formatter = null;
//   if (simpleFormatter != null) {
//       formatter = (DateFormat)simpleFormatter.get();
//   }
//   if (formatter == null) {
//       /* No cache yet, or cached formatter GC'd */
//       formatter =
//         new SimpleDateFormat("EEE MMM dd HH:mm:ss zzz yyyy,
//             Locale.US); //Note standard format defined here!
//       simpleFormatter = new SoftReference(formatter);
//   }
//        synchronized (formatter) {
//           formatter.setTimeZone(TimeZone.getDefault());
//           return formatter.format(this);
//        }
//    }//toString()
```

In this example the block insertion of // at the beginning of each line effectively removes the entire method from the file. The user who commented out this method did not need to be concerned that there might be existing comments of /* .. */ style or // style. While there are such existing comments, they are not affected. If the method is ever reinstated the simple removal of the leading // from each line will return the method intact.

STY-19 Properly format comments.

A comment can be thought of as the 'title' of a line or paragraph of code. As such it should proceed the code but not be separate from it. This also means it should have the same indenting and max line width as the code section. By the same token it should be separate from the preceding section.

Therefore, characteristics of properly formatted comments are:

1. A comment block should be preceded by a single blank line.
2. A comment should precede the code to which it relates.
3. A comment should be indented to the same level as the code it relates to.
4. End of line comments should be offset from the code so that the comment stands out.

STY-20 Comments should not obscure the code

If you have an algorithm or strategy which requires extensive explanation in comments consider a large comment block which precedes the described code. Use this form rather than sprinkling comments throughout the code if there will be so many comments that the structure of the code will be difficult to see.

Example:

If you have complex code you have the following choices, with later choices being better:

A) No comments at all.

```
double newSpeed = m_currentAngle - newAngle;
double newAccel = m_Speed - newSpeed;
if(newAccel > PI)
{
  newAccel = newAccel - TWOPI ;
}
else if(newAccel < -PI)
{
  newAccel = TWOPI - newAccel;
}
```

B) Sprinkle comments throughout the algorithm, possibly obscuring the code

```
double newSpeed = m_currentAngle - newAngle;
double newAccel = m_Speed - newSpeed;

// see if we've wrapped around the circle
if(newAccel > PI)
{
  // go the other way around the circle
  newAccel = newAccel - TWOPI ;
}
else if(newAccel < -PI)
{
  // go the other way around the circle
  newAccel = TWOPI - newAccel;
}
```

C) Create a comment before a block of algorithmic code

```
/* determine the speed and the accelerations required
 * to point there then see if we're wrapped around
 * the circle and adjust accordingly
*/
double newSpeed = m_currentAngle - newAngle;
double newAccel = m_Speed - newSpeed;

if(newAccel > PI)
{
  newAccel = newAccel - TWOPI ;
}
else if(newAccel < -PI)
{
  newAccel = TWOPI - newAccel;
}
```

D) Refactor the code to an appropriately named routine which you just call

```
double getAdjustedAcceleration(double oldAngle, double newAngle)
{
  double newSpeed = m_ oldAngle - newAngle;
  double newAccel = m_Speed - newSpeed;


   if(newAccel > PI)
  {
    newAccel = newAccel - TWOPI ;
  }
  else if(newAccel < -PI)
  {
    newAccel = TWOPI - newAccel;
  }
 return(newAccel);
}// getAdjustedAcceleration
```

STY-21 Variable declaration grouping.

Variables declarations should have the following characteristics:

1. One declaration per line. This encourages commenting of each variable.
2. Order the declarations in some fashion. A common ordering is alphabetic by type first, then by variable name. However, in some cases dependencies can require a different order.
3. New declarations should go into their appropriate place according to the ordering being used. A comment should indicate when and why the new variable was added.

STY-22 Place variable declarations at the beginning of the innermost enclosing block.

Java allows variables to be declared immediately before they are needed. However, rather than placing the immediately before first use, declarations should be placed at the beginning of the innermost block in which they are used. This convention provides the reader with known positions to locate declarations. Without it, the reader must scour the code to find the declaration.

One example of variable abuse is the re-use of a variable, for a completely different purpose, just because it has already been declared and is still in scope. One of the best ways of limiting such abuse of variables is to limit their existence to the innermost block which requires them.

To summarize: Class variables should be declared at the top of the class, method variables at the top of the method, block variables at the top of the block.

The only exception is the declaration of for-loop index variables which can be declared in the for statement, itself. This is useful since it automatically limits the scope of the index variable to the for loop, preventing the inadvertent use of the index after the loop terminates.

STY-23 Limit the number of Java statements per line to 1.

Multiple statements per line can hide code to the casual observer. Also, it limits the ability to step through the code statement by statement. Since we limit declarations to one per line, following the principle of uniformity, you should also limit statements in a similar manner. See [Conventions] 7.1

Example:

```
Public double sumAllBids(Collection bids) {
  double result = 0;
  for (Iterator iter = bid.Iterator(); iter.hasNext(); ) { result +=
  calcTotal((BidRecord)iter.next());if (result > 10000) {break;}}
  return result;
}
```

The example above should compile, but will take a significant amount of effort to figure out exactly what the sumAllBids() method is really doing.

STY-24 Optional braces are not optional.

Compound statements use braces to delimit one or more statements under the control of the compound statement. These braces should always be used. This is true, even if the braces surround only one statement and would be optional in that instance. Omitting the braces can lead to errors since it allows an additional statement to be added. Although this additional statement appears to be under the control structure, in fact, it is not. See [Conventions] 7.2 and 7.4.

Two different schools of thought exist on whether extra braces should be used. On one hand the use of braces is the key indicator that multiple statements should be treated as one. Following this logic, a single statement would never have a set of braces around it.

On the other hand using braces wherever they are allowed provides for a more uniform coding style. It also sets up the code to easily accommodate multiple statements when a single statement is grown into more than one. The failure to add braces when a single statement is changed into multiple statements is a common error which is avoided when optional braces are always used.

Example 1:

```
if (Character.isLetter(ch))
{
  flag = 1;
}
if (Character.isDigit(ch))
{
  flag = 2;
}
if (Character.isSpaceChar(ch))
```

```
{
  flag = 3;
}
```

The above example is preferable over the following one:

```
int flag=0;
if (Character.isLetter(ch))
  flag = 1;
if (Character.isDigit(ch))
  flag = 2;
if (Character.isSpaceChar(ch))
  flag = 3;
```

Example 2a: Using braces

```
public double sumAllBids(Collection bids) {
  double result = 0;
  for (Iterator iter = bid.Iterator(); iter.hasNext(); ) {
    result += calcTotal((BidRecord)iter.next());
  }
  return result;
}
```

Example 2b: Not using braces

```
public double sumAllBids(Collection bids) {
  double result = 0;
  for (Iterator iter = bid.Iterator(); iter.hasNext(); )
    result += calcTotal((BidRecord)iter.next());
  return result;
}
```

In the Example 2a, it is clearly evident where the for loop begins and ends. In addition, a very common bug introduced when enhancing code is to forget to add the surrounding brace as is shown in Example 2c below.

Example 2c: Bug caused by forgetting braces

```
Public double sumAllBids(Collection bids) {
  double result = 0;
  for (Iterator iter = bid.Iterator(); iter.hasNext(); )
    result += calcTotal((BidRecord)iter.next());
    if (result > 10000) {
      break;
    }
  return result;
}
```

Example 2c has two problems: (1) inconsistent coding style, and (2) the for loop is missing the bounding braces. It is difficult for the reader to see why the loop does not end after the maximum total of 10,000 has been reached.

STY-25 Parameter naming.

Name each method parameter based on the role which it provides. Be descriptive. Clearly named variables are a critical success factor in avoiding cryptic code which is difficult to use and maintain.

Consider the following two examples:

```
public double calcAvgPrice1(double b, double a) {
  return b/a;
}

public double calcAvgPrice2(double totalPrice, double units) {
  return totalPrice/units;
}
```

Both examples above produce the correct result. However, it is much clearer for someone calling calcAvgPrice2() to know exactly what value should be passed for each parameter.

STY-26 Method naming for accessor methods.

Names of methods which function as accessors or a manipulators of private variables should follow the JavaBean convention getX() or setX().

For example:

```
class ValueObject {
  private Integer intValue;

  public Integer getIntValue() {
    return intValue;
  }
  public void setIntValue(Integer a) {
    intValue = a;
  }
}
```

In the example above, the data member intValue is encapsulated by the methods getIntValue() and setIntValue(). Note that the lower case first letter of the data member variable name is converted to upper case when naming the methods.

STY-27 Use prefixes to indicate variable scope and source.

The overall scope of a variable can be more important than knowing its type so use prefixes to indicate where and how the variable was declared. This is a simple, easy, and non-intrusive way to designate method parameters, local variables, and class variables.

Sample scheme1:

```
int m_internalSpeed;    //Class-level member variables use
                        //an “m_” prefix.
int l_loopIndex;        //Variables local to the method are
                        //prefixed with “l_”.
int p_opCode;           //Method parameters get a “p_” prefix.
```

Sample scheme2:

```
int gInternalSpeed; //Public, class-level variables get
                        //an “g” prefix (for "global").
int fInternalSpeed;     //Private, class-level variables use
                        //an “f” prefix (for "field").
int vLoopIndex;         //Variables local to the method are
                        //prefixed with “v”.
int pOpCode;            //Method parameters get a “p” prefix.
```

Sample scheme 3 (where only class-level variables are distinguished):

```
int m_internalSpeed;    //Class-level member variables use
                        //an “m_” prefix.
```

Sample scheme 4:

Use no prefixes at all. If methods are short, which they should be, the code for the method will be completely viewable as a unit and use of prefixes is not as useful.

Remember, consistency is more important than which style is selected. Pick one style and stick with it.

STY-28 Use blank lines to organize code blocks.

Blank lines should be used to group code. The larger the construct the more spacing should be used to offset it. For example, spacing between methods in a class should be smaller than between classes in a file.

Single blank lines should be used:

1. Between local variable declarations and the first code in a method.
2. Before a block comment.
3. Between logical sections of code to improve readability.

Double blank lines should be used:

1. Between methods
2. Between class and interface definitions.
3. Between any other sections of a source file.

See [Conventions] 8.1

STY-29 Name all constants and define them in one location only.

Avoid embedding magic values in the code. Instead, define them in a central location and use a named reference in the code. Exceptions are -1, 0, and 1, which might be needed for loop control and testing boundary conditions. (See [Conventions] 10.3)

When creating a file containing application constants, make the file an interface rather than a class.

Improper way to handle constant values:

```
public class MyClass
{
    public void myMethod1()
    {
        if(t1.equals("test string 1"))
        {
            ....
        }
        else if(t1.equals("test string 2"))
        {
            ....
         }
    }
}
```

Proper way to handle constant values:

```
interface Test
{
  public static string CONDITION1 = “test string 1";
  public static string CONDITION1 = “test string 2";
}

public class MyClass implements Test
{
  public void myMethod1()
  {
    …
    if(t1.equals(Test.CONDITION1))
    {
      …
```

```
    }
    else if(t1.equals(Test.CONDITION2))
    {
      …
    }
  }
}
```

See STD-5 for naming of constants.

Conventions

CON-1 Protect code using try..finally.

Try..finally should be used more than it typically is. Use finally after try..catch statements to ensure execution of important code. For example, when opening a stream or database connection in a method, use a try..finally structure to ensure the stream or connection close method is always called. Assume the worst. Code defensively.

Example:

Here the database connections, statements, and result sets are guarded by try..catch so they will all be closed.

```
import java.sql.*;
import java.util.*;
/**
 * Database Accessor class
 */
public class DbUsers
{
  /**
  * Gets database Users of the project xxxx
  */
  public Vector getUserNames() throws SQLException
  {
    Connection conn=null;
    Statement stmt = null;
    ResultSet rs=null;
    Vector names=null;
    try
    {
      conn = getConnection();
      stmt = conn.createStatement();
      rs = stmt.executeQuery("select name from db_users");
      while(rs.next())
      {
        names.addElement(rs.getString(1));
      }//while
    }//try
    finally
    {
      if(conn != null) conn.close();
      if(stmt != null) stmt.close();
      if(rs != null) rs.close();
      return names;
    }//finally
  }//getUserNames()

  /**
  * gets Database connection
```

```
  */
  public Connection getConnection() throws SQLException
  {
    try
    {
      Driver dr =
          (Driver)Class.forName("wl.jdbc20.pool.Driver").newInstance();
      return dr.connect("jdbc20:wl:pool:esmmPool", null);
    }
    catch (InstantiationException ie)
    {
      return null;
    }
    catch (ClassNotFoundException cl)
    {
      return null;
    }
    catch (IllegalAccessException il)
    {
      return null;
    }
  }//getConnection()
}//DbUsers
```

CON-2 Constructors must leave objects in a stable state.

Constructors should be complete enough so that subsequent calling of other methods immediately after construction should not fail in unexpected ways. In other words, once an object is constructed, it should be well-behaved even if it is not in a useful state.

Example:

```
class Purchase {
   static int noOfPurchases;
   Purchase() {
      noOfPurchases++;
   }
}

class LiqPurchase extends Purchase {
   LiqPurchase(int age, String day) {
     if(age < 21 || day.equals("Sunday")){
        System.out.println("No liquor purchase possible");
     }
   }

   public static void main(String args[]) {
      Purchase p1= new LiqPurchase(30,"Monday");
      Purchase p2= new LiqPurchase(18,"Monday");
      //The number of purchases will printed as 2 instead of
      // 1, since the second time there was no purchase made.
      System.out.println("Number of purchases: " +
        " " + p2.noOfPurchases);
```

```
    }
}
```

CON-3 Methods should accomplish a single task.

Avoid creating long processing sequences which can be logically divided into separate methods. If you find yourself naming a method with “And” in the name, consider this a clue that you may have included too much functionality in one method.

Example:

Here something that could have been placed inline has been divided into four separate methods, each contributing their part to the overall purchase order.

```
public String getPurchaseOrderXml()
{
  String po =
     getXmlHeader() +
     getPoHeader() +
     getLineItemDetails() +
     getPoSummary();
  return po;
}
```

CON-4 Limit the length of methods.

A method should rarely exceed a “page” of code. This helps ensure it accomplishes a single function and enhances its readability.

A page would normally be considered 30 lines or so. Pick a specific limit and stick to it.

CON-5 Limit the length of source files.

While you can include an entire package in one source file, that file may grow to be really long. Source files should have a maximum length, given in lines, which is used as a rule for deciding when sections should be split into separate files. Although the maximum acceptable length of a source file is open to discussion, a good working limit is about 2000 lines.

If a source file exceeds this limit due to the inclusion of multiple classes, then move selected classes into another file. Separating each public class into a separate file makes sense. If a single class exceeds this limit, consider decomposing it into a set of smaller classes.

CON-6 Limit the number of methods.

Classes which provide services to callers can quickly become ungainly and difficult to maintain. To keep classes focused on their primary tasks, limit the number of methods. While the maximum acceptable method count is open to discussion, a good guideline is 20 (including accessor methods).

One manner in which this count can become exceeded is after refactoring. For example, if you combine two classes into one you may end up with an excessive number of methods in the resulting class. Therefore, you must balance the benefit of refactoring with the need to keep the method count low.

CON-7 Limit the use of public methods and variables.

Because the public designation implies that the method or variable's usage outside the class is required for proper use of the class, only make public that which needs to be public. Remember that the reader will be trying to discern the proper use of the class from its public interface. Having non-essential elements in that interface will only be confusing. Never designate a public element on the off chance it might need to be public later. Instead, return to the class and change the visibility of the element when its purpose changes. And, oh yes, document the change in the JavaDoc comments by explaining why the element became public.

See [Conventions] 10.1

Example:

This class has rightfully designated getPrice, getTotal, and getDiscount methods. However, the getDiscount and getPrice methods remain private until there is a demonstrated need for them to be public. The private getTotal() method provides a total for any given quantity, while the public getTotal() method provides the final total amount.

```
class Summary
{
  private double getPrice()
  {
    …
  }

  private double getTotal(int quantity)
  {
    …
  }

  private double getDiscount(float price)
  {
    …
  }
```

```
    public double getTotal()
    {
      return getTotal(quantity) * getPrice() - getDiscount(price);
    }
  }
```

CON-8 All class variables should be private.

Avoid the use of public variables. If public access is needed, then use accessor methods. Once made public, exported variables become part of the interface contract of the class, so designating variables as public locks you into *always* providing these variables. Using accessor methods allows you to change the internal implementation of the variables, if this becomes necessary, without changing the interface contract. Code to retain future flexibility.

Example:

```
// The following class implements the
// database connection pool.
Public ConnectionPool
{
   public Vector connectionPool ;
   public String url;
   public String username;
   public String password;
   public int poolSize;

   public ConnectionPool(String url, String username, String password, String poolSize)
   {
     …
   }

   public synchronized Connection getConnection()
   {
     …
   }

   public synchronized void removeConnection(Connection connection)
   {
     …
   }

   public synchronized void closeConnection()
   {
     …
   }
}
```

All variables can be accessed and reassigned directly since they are public. However, allowing an external source to read user name and password will introduce a security problem.

Also, PoolSize is crucial. If it is accidentally or deliberately set to zero, the application will stop running since the database connection is no longer available. Therefore, it should be private and set only through an accessary method which can detect problems and avoid setting Pool size to an invalid value.

CON-9 Limit the number of parameters.

Method signatures present an interface to the caller which is more difficult to use as the number of parameters grows. While the maximum acceptable parameter count is open to discussion, a good rule of thumb is to limit parameters to 5.

If more parameters are needed, switch to using accessary methods or pass an instance of a helper class which carries a complete set of values.

Example:

```
Public class BankAccount
{
    public BankAccount(String name, String streetAddress, String city,
        String state, int zipCode, int ssn, Date dob,
        float initialBalance, String phone, String email)
}
```

We can use one of the following alternatives:

```
Public class BankAccount
{
   public BankAccount(String name, int ssn, Date dob, float initialBalance)
}
```

Then call setX method to pass streetAddress, city, state, zipCode, phone and email.

Alternatively we can create two classes: HelperAddress and HelperContact. HelperAddress class will include street name, city, state and zip code. Contact class contains phone number and email. This leaves the constructor with the remaining parameters:

```
Public class BankAccount
{
   public BankAccount(String name, int ssn, Date dob, float initialBalance)
}
```

Then calls to setHelperAddress and setHelperContact associate those objects with the bank account.

For more information see “Introduce Parameter Object” in [Fowler99].

CON-10 Avoid predefined shallow classes and methods.

Do not implement classes, methods or class data members, which are not immediately useful. Introduce them later when a subsequent release is ready to be used. Until that point they are clutter.

CON-11 Include proper content in your implementation comments.

STY-19 describes the formatting of comments. This convention addresses what to put into those comments.

Implementation comments should first explain why the code is written as it is. The intentions of the author should be spelled out so the reader does not have to guess them or otherwise infer them from the code. This way the reader can use any discrepancy between the intention of the author and the reality of the code to help diagnose a problem.

In other words, do not merely describe how the code is supposed to work but why you implemented the design that way.

Example:

```
/**
* The method reads a XML document defined
* in a file given by users and returns a
* composed XML string. A string buffer class is
* used because the string will be changed as
* more characters are read from the file.
* BufferedReader is the best choice in this case.
* It will provide for the efficient reading of lines.
*
*@ parameter  filename File
*@return java.lang.String
*/

Public String getXMLString(File fileName)
{
    StringBuffer sb = new StringBuffer();
  try {
          BufferedReader br =
            new BufferedReader(new FileReader(fileName));

          boolean finished = false;
          while(!finished) {
             String str = br.readLine();
             if(str !=null) {
                sb.append(str);
             }else{
                finished = true;
             }
          }
       }catch (Exception ex) {
```

```
            throw ex;
        }
      return sb.toString();
}
```

Since implementation comments should focus on how the code is written to achieve the desired result, the above example properly explains some important details. In contrast, the following example simply states what the method does.

```
/**
* Read an XML string from a given file and return the string.
*
*@ parameter  filename File
*@return java.lang.String
*/

Public String getXMLString(File fileName)
{
  …
}
```

One further point: Use specific flag strings to make note of coding practices which are questionable but work. Use such comments to explain that you understand and acknowledge the issues with the code. One suggested flag is “Note:” as in:

```
//Note: This sort method is slow but cheap to
// implement correctly. If the
// size of the array grows too large
// another method will be needed.
```

Use another specific flag string to highlight broken code which must be fixed later. One suggested flag is “FIXME:”.

See [Conventions] 10.5.4

CON-12 Avoid nesting conditions more than 3 deep.

Rewrite the conditions if more seem to be needed.

For example, this code nests more than 3-levels:

```
String url = null;
If(propertyList != null)
{
  for(int i=0;i<propertyList.size(); i++)
  {
    Properties prop = (Properties)propertyList.elementAt(i);
    if(obj != null)
    {
      If(prop.containsKey("url"))
```

```
        {
          url = prop.getProperty("url");
          break;
        }
      }
    }
  }
```

This can be re-written using exception catching as:

```
String url = null;
try
{
  for(int i=0;i<propertyList.size(); i++)
  {
    Properties prop = (Properties)propertyList.elementAt(i);
    if(prop.containsKey("url"))
    {
      url = prop.getProperty("url");
      break;
    }
  }
}
catch (NullPointerException ex)
{
  throw ex;
}
```

CON-13 Define constants in interfaces.

When writing an application, place all application-related constants in a single application interface rather than spreading them throughout other interfaces or classes.

Any Java class or interface in the application can then gain access to the constants by simply implementing the application-specific interface. This technique also has the advantage that all application constants are defined in a single location, making them easy to find and maintain.

It is important to note here that this convention refers to true applications constants, and not those variables that have the possibility of changing or being modified. Dynamic variables should be contained in XML based property files or a similar data store.

Example:

```
public interface ApplicationConstants
{
  public static int NEW_ORDER_REQUEST = 1;
  public static int CANCEL_REQUEST = 2;
  public static int PAYMENT_REQUEST = 3;
}
```

```
public class SomeClass implements ApplicationConstants
{

  public void processRequest(int action, RequestData data)
       throws InvalidRequestException
  {
    if (action == NEW_ORDER_REQUEST)
      processNewOrder(data);
    else if (action == CANCEL_REQUEST)
      cancelOrder(data);
    else if (action == PAYMENT_REQUEST)
      processPayment(data);
    else
      throw new InvalidRequestException(action);
  }

}
```

CON-14 Make good use of spacing.

Blank spaces can be an important formatting tool by visually associating related code and separating non-related code.

1. When a keyword is followed by a parenthesis, the two should be separated with a single space. For instance, type casts should always be followed by a space. However, a method name should not be separated from its opening parenthesis. This helps the reader recognize method declarations and distinguish them from type casts.
2. A blank space should appear after a comma in a list but *not* before a comma. Same with the expressions in a for statement - one space following each semi-colon.
3. Binary operators (except .) should be separated from their operands with a space. However, if an expression becomes too long, this can be relaxed. When removing spaces to shorten an expression, remove them from the higher precedence operators first. This results in visually identifying the operations which are performed first.
4. Unary operators should not be separated from their operands with any spaces.

See [Conventions] 8.2

CON-15 Class methods and class variables should only be accessed via the class identifier.

Never use an instance identifier to access a class variable or method- use the class name identifier instead. This highlights the nature of the class variable or method.

See [Conventions] 10.2

CON-16 Use parentheses to clarify expressions.

Even if the default operator precedence is correct, adding parentheses can still clarify an expression by emphasizing the fact that a certain precedence is required by the code.

See [Conventions] 10.5.1

This is especially true when using the ternary operator “?”.

For example

```
x >= 0 ? x : -x;
```

should be written

```
(x >= 0) ? x : -x;
```

See [Conventions] 10.5.3

CON-17 Identify closing braces.

Adding a // comment immediately after a closing brace for a class or method helps to identify the end of the declaration. These can be valuable markers when navigating the code, especially when the entire class or method is not visible all at once.

Example:

See example for CON-22.

Some readers may have noted that we have not done this is all our examples in this document. Since most of the examples are short we decided this step was not necessary. This is a good example of balancing the need for properly identifying closing braces vs. the desire not to clutter the code.

CON-18 Favor IsX() over getX() or hasX() for boolean functions.

For methods returning boolean use names like isX rather than getX or hasX.

CON-19 Avoid sub-classing the class Error.

Leave Error and sub-classes of Error for use by the JVM to indicate resource deficiencies, invariant failures, or other conditions which make it impossible to continue execution.

CON-20 Distinguish between checked and unchecked exceptions.

Use checked exceptions for conditions from which the caller can reasonably be expected to recover. Use runtime exceptions for programming errors. See Item 40 in [Bloch01].

Likewise, unchecked exceptions (runtime exceptions) should *not* be included in throws clauses but *should* be included in JavaDoc details via the @throws tagged comments.

CON-21 Enhance exceptions with additional data.

Provide additional data members or accessary methods in exceptions you define so that callers do not need to parse exception strings to determine details.

CON-22 Avoid dependency on side-effects.

Due to short-circuiting of expression evaluation, side-effects will not occur in expressions that are not executed.

Example:

```
class CountPurchase {
      int hour;
      static int noOfPurchase=0;

      CountPurchase(int h) {
        hour = h;
        changeCashier();
        System.out.println("The number of Purchase is: " + noOfPurchase);
      }//CountPurchase()

     void changeCashier() {
        if(hour > 17 || noOfPurchase++ > 40) {
            System.out.println("Time to change the cashier");
        }
     }//changeCashier()

      public static void main(String args[]) {
          CountPurchase c1 = new CountPurchase(15);
          //The increment operation will not be performed
          // the second time and the number of purchases
          // printed will be 1 instead of 2.
          CountPurchase c2 = new CountPurchase(20);

     }//main()
}//CountPurchase
```

References

We have used several previous works as references. These are:

[Ambler00] Ambler, Scott W. *Writing Robust Java Code - The AmbySoft Inc, Coding Standards for Java v17.01d*

http://www.AmbySoft.com/JavaCodingStandards.pdf

A wealth of good advice is contained in this document.

[Bloch01] Bloch, Joshua. *Effective Java Programming Language Guide*, Addison-Wesley, Reading, MA, 2001. ISBN: 0-201-31005-8

[Conventions] *Code Conventions for the Java™ Programming Language,* Sun Microsystems.

http://java.sun.com/docs/codeconv/

This is the document we used as a starting point in our discussions.

[Fowler99] Fowler, Martin. *Refactoring, Improving the Design of Existing Code*, Addison-Wesley, 1999. ISBN 0-201-48567-2

[Haggar00] Haggar, Peter. *Practical Java Programming Language Guide,* Addison-Wesley, Reading, MA, 2000. ISBN: 0-201-61646-7

Several recommendations made by Peter Haggar in this book are echoed in our list.

[JLS] Gosling, James, Bill Joy, Guy Steele, Gilad Bracha. *The Java™ Language Specification, Second Edition*, Addison-Wesley, Boston, 2000. ISBN: 0-201-31008-2.

List of Standards

List of Styles

List of Conventions

Index