NATIONAL WEATHER SERVICE
OFFICE of HYDROLOGIC DEVELOPMENT

Science Infusion Software Engineering Group (SISEPG)
JAVA Programming Standards and Guidelines

Version 2.0

Revision History

Date	Version	Description	Author
03/30/2007	1.9	Initial Version	SISEPG
03/28/2008	2.0	Introduced Allman style of bracing. The "One True Bracing Style (OTBS) is now correctly linked to Kernighan and Ritchie style of bracing. Added section 2.5	SISEPG

1 Introduction

The Office of Hydrologic Development (OHD) develops and maintains software which the National Weather Service (NWS) Weather Forecast Offices (WFOs) and River Forecast Centers (RFCs) use to generate hydrologic forecasts and warnings for rivers and streams across the country. OHD also develops and maintains software which runs centrally to acquire and distribute critical data to the WFOs and the RFCs. Software development and maintenance has become a critical component supporting the operations of NWS forecast offices and it is essential that it be well written and maintained.

Well written software offers many advantages. It will contain fewer bugs and will run more efficiently than poorly written programs. Since software has a life cycle and much of which revolves around maintenance, it will be easier for the original developer(s) and future keepers of the code to maintain and modify the software as needed. This will lead to increased productivity of the developer(s).

The OHD Science Infusion Software Engineering Process Group (SISEPG) has developed standards and guidelines to ensure that developers follow good and widely accepted software development practices when coding. It is believed this will lead to well written and better structured programs, which must be simple, intuitive, and uniform. The overall cost of the software is greatly reduced when the code is developed and maintained according to software standards.

This document will present standards and guidelines for the Java Programming Language. The Java standards are programming techniques which OHD developers are expected to follow to assist them in writing high quality software. The Java standards will be enforced through peer reviews and code walkthroughs. The Java guidelines are good programming practices that developers are encouraged to adopt.

The developer should also read the OHD General Software Development Standards and Guidelines document to become familiar with the standards and guidelines deemed by the SISEPG to be applicable to all programming languages.

It is important to note that standards are not fixed, but will evolve over time. Developers are encouraged to provide feedback to the SISEPG (sisepg@gateway2.nws.noaa.gov). Also each project area may derive its own standards and guidelines if special requirements are desired. Finally, the developers are to follow the "OHD General Software Development Standards and Guidelines" except where specified in this document.

.

1.1 Acknowledgments

This document follows a very similar format to the Java Code Conventions document from Sun Microsystems. Much of the text has been taken word for word from that document, but some of it was altered to suit the needs of the OHD. The idea is to use most of the recommended conventions by Sun Microsystems. This will better enable future Java programmers in writing and maintaining OHD code.

2 Standards

2.1 File Names

File Type	Suffix
Java Source	.java
Java Bytecode	.class

2.3 Indentation

Three or four spaces shall be used as the unit for indentation and whichever is used, the developer shall be consistent. **Do not use tabs**.

2.3 Braces {}

There are two acceptable styles of bracing. The first and preferred style is known as the "Allman" Style". The "Allman" style places the open brace '{' on the it's own line and before a block of code. The second is the older Kernighan and Ritchie style (a.k.a "One True Bracing Style or OTBS") which places the open brace on the same line of the statement that begins a block of code. Both styles will be show in this section, but only the "Allman" will be shown in examples in subsequent sections of this document.

Examples of the preferred "Allman":

```
if ( condition1 )
{
    doSomethingAboutIt();
}

if ( condition1 )
{
    doThis();
}
else
{
    doThat();
}


for ( i = 0; i < 10; i++ )
{
    initializeThese();
}
```

```
while ( i < 10 )
{
    initializeThese();
}

do
{
    initializeThese();
}
while ( i < 10 );
```

Examples of Kernighan and Ritchie Style (or OTBS):

```
 if ( condition1 ) {
     doSomethingAboutIt();
 }

 if ( condition1 ) {
     doThis();
 }
 else {
     doThat();
 }


for ( i = 0; i < 10; i++ ) {
    initializeThese();
}

while ( i < 10 ) {
    initializeThese();
}

do {
    initializeThese();
} while ( i < 10 );
```

Although the “Allman” style is preferred, consult the project leader to verify the bracing style being used on a particular project.

Whichever bracing style is used, the developer must be consistent throughout a program. When editing existing code, the developer must adopt the bracing style already in use.

2.4 File Organization

A file consists of sections that should be separated by blank lines and optional comments identifying each section. A section is considered to be any of the following: beginning comments, Package and Import Statements, and Class and Interface Declarations (which include variable declarations, constructors, and methods).

Files longer than 2000 lines are cumbersome and should be avoided.

2.4.1 Java Source Files

Each Java source file contains a single public class or interface. When private classes

and interfaces are associated with a public class, the developer can put them in the same source file as the public class. The public class should be the first class or interface in the file.

Java source files have the following order:

a. Beginning comments should use OHD standards which will apply to all languages, though Javadoc comment syntax will be used.
b. Package and import statements.
c. Class and interface declarations.

2.4.2 Package and Import Statements

The first non-comment line of most Java source files is a package statement. After that comes the import statement. For example:

```
package java.awt;

import java.awk.per.CanvasPeer;
```

2.4.3 Class and Interface Declarations

The class or interface declaration should be in the following order:

1. Class/interface documentation comment (/**...*/)

   NOTE: For more information see "How to Write Doc Comments for Javadoc" at

   http://java.sun.com/products/jdk/javadoc/writingdoccomments.html.

   For more information about doc comments and java doc:

   http://java.sun.com/products/jdk/javadoc

2. Class or interface statement

3. Class/interface implementation comment (/*...*/), if necessary.

   NOTE: This comment should contain any class-wide or interface-wide information that was not appropriate for the class/interface documentation comment.

4. Class (static) variables

   Declared in the following order:
   - the `public` class variables
   - the `protected` class variables
   - the package level (no access modifier)
   - the `private` class variables.

5. Instance variables

   Declared in the following order:
   - the `public` instance variables
   - the `protected` instance variables
   - the package level (no access modifier)
   - the `private` instance variables.

6. Constructors

7. Methods

   NOTE: These methods should be grouped by functionality rather than by scope or by accessibility. For example, a private class method can be in between two public instance methods. The goal is to make reading and understanding the code easier.

2.5 Using Doc Comments to Create Documentation Block

The General Standards and Guidelines state that each file should have a documentation block as well as each module within the file. In Java, this translates to each Java class and interface should have a documentation block as well as each method. Rather than using the typical comment (block, single line) for this documentation block, the developer shall use 'doc' comments. This allows the current or any future developer to run the Javadoc program to produce API documentation in HTML format.

The next couple of sections will define what is required to be documented. It is important to note brackets are used to denote what the user is to enter. The brackets are not to actually appear in the 'doc' comments. It is also important to note that some block tags, such as @param, @author, @exception may have multiple entries. Finally there are more block tags that are available (see http://java.sun.com/j2se/javadoc/writingdoccomments/), but if they don't appear in the next couple sections, they are considered optional and should be used as needed.

2.5.1 Class or Interface Doc Comments

All classes and interfaces shall have the following preceding the class or interface:

```
/**
 * [Description of the class/interface]
 * @author [name]
 */
```

2.5.2 Method Doc Comments

All methods, with the exception of well-named "getters" and "setters" shall have the following doc comments:

```
/**
 * [Description of method, including any files or database tables
 *  which may be required by the code]
 * @param [name] [description of parameter passed]
 * @return [what] [description of what is return and possibly why]
 * @throws [exception type]
 */
```

NOTE: Omit @return for methods that return void and for constructors. Omit @throws if method doesn't throw any exceptions.

Well-named "getters" and "setters" can be subjective, but listed below are few examples of some well-named and poorly named "getters" and "setters".

Examples of well-named "getters" and "setters":

void setMaximumAllowedComponents(int maxComponentCount);
int getMaximumAllowedComponents();
String getPhysicalElementCode();

Examples of poorly-named "getters" and "setters":

setMax(); //max of what ?
setMaximum(); //maximum of what?
getMax(); //max of what?

setCount(int count); //count of what?
int getCount();

String getCode(); //what kind of code are we talking about?

3 Guidelines

3.1 Line Length

When practical, avoid using lines longer than 80 characters since they are not handled well by many terminals and tools. When the line length must exceed 80 characters, be sure it does not exceed 120 characters. Examples for use in documentation should have a shorter length, with no more than 70 characters.

3.2 Wrapping Lines

When an expression does not fit on a single line, it should be broken according to these general principles:

- Break after a comma.
- Break after an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level as the previous line.
- If the above rules lead to confusing code or to code that is squished up against the margin, indent 8 spaces instead.

Here are some examples of breaking method calls:

```
someMethod (longExpression1, longExpression2, longExpression3,
            longExpression4, longExpression5);

var = someMethod1 (longExpression1,
                   someMethod2(longExpression2,longExpression3) );
```

The following examples are of breaking an arithmetic expression. The first is preferred, since the break occurs outside the parenthesized expression, which is at a higher level.

Prefer:

```
longName1 = longName2 * (longName3 + longName4 - longName5) +
            4 * longName6;
```

```
Avoid:

longName1 = longName2 * (longName3 + longName4 -
            longName5) + 4 * longName6;
```

The following are two examples of indenting method declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used convention indentation. Therefore indent only 8 spaces.

```
// CONVENTIONAL INDENTATION

someMethod (int anArg, Object anotherArg, String yetAnotherArg,
```

```
            Object andStillAnother)
{
...
}

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS

private static synchronized aLongMethodName (int anArg,
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother)
{
...
}
```

When line wrapping occurs with **if** statements and the bracing style used is Kernighan and Ritchie, ensure the body of the if-statement is easily seen. For example:

```
//NOT SO GOOD

if ((condition1 && condition2) ||
    (condition3 && condition4) ||
    !(condition5 && condition6)) {
    doSomethingAboutIt ();         //MAKES THIS LINE EASY TO MISS
}
```

```
//INSERTING A BLANK LINE AFTER THE IF MAKES THE
//BODY OF THE IF MORE EASILY SEEN

if ((condition1 && condition2) ||
    (condition3 && condition4) ||
    !(condition5 && condition6)) {

    doSomethingAboutIt ()
}
```

When using the "Allman" style of bracing, this same line is easier to see. This is an advantage of using the "Allman" style.

```
//SAME CODE USING THE ONE-TRUE-BRACING-STYLE

if ((condition1 && condition2) ||
    (condition3 && condition4) ||
    !(condition5 && condition6))
{
    doSomethingAboutIt ();
}
```

3.3 Comments

Java programs can have two kinds of comments: 1) implementation comments; and 2) documentation comments. Implementation comments are those found in C and C++, which are delimited by /*...*/, and //. Documentation comments (known as doc comments) are Java-only, and are delimited by /**...*/. Doc comments can be extracted to HTML files using the javadoc tool.

Implementation comments are meant for commenting out code or for comments about the particular implementation. Doc comments are meant to describe the specification of the code, from an implementation-free perspective to be read by developers who might not necessarily have the source code at hand.

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment.

Discussion of nontrivial or non obvious design decisions is appropriate, duplicating information that is present in (and clear from) the code should be avoided. It is too easy for redundant comments to get out of date. **In general, avoid any comments that are likely to get out of date as the code evolves**.

Programs can have four styles of implementation comments: block, single-line, trailing and end-of-line.

3.3.1 Block Comments

Block comments are used to provide descriptions of files, methods, data structures, and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code.

```
/*
 * Here is a block comment.
 */
```

Block comments may also use // at the beginning of the lines. Example:

```
// Here is the first line of a block comment.
// Here is the second line.
```

Block comments can start with /*-, which is recognized by **indent (1)** as the beginning of a block comment that should not be reformatted. Example:

```
/*-
 * Here is a block comment with some very special
 * formatting that I want indent(1) to ignore.
 *
 * one
 *     two
 *         three
 */
```

Note: If you do not use **indent (1)**, you do not have to use /*- in your code or make any other concessions to the possibility that someone else might run **indent(1)** on your code.

3.3.2 Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment cannot be written in a single line, it should follow the block comment format (see section 3.3.1). A single-line comment should be preceded by a blank line. Here is an example of a single-line comment in Java code:

```
if (condition)
{

    /* Handle the condition. */
    ...
    // Or you can use this format for a single line comment.

}
```

3.3.3 Trailing Comments

Very short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same column.

Here is an example of a trailing comment in Java code:

```
if ( (a % 2) == 2 )
{
    return TRUE;          /* special case */
}
else
{
    return isPrime(a);    /* works only for odd a */
}
```

3.3.4 End-Of-Line Comments

The // comment delimiter can comment out a complete line or only a partial line.
The following example shows complete line comments as well as partial line comments.

```
// This code has been commented out and replaced with the code
// code to follow. This code can be deleted is 3 months if still
// not applicable.
//
//if (a == 0)
//{
//    doThis();
//}

if ( (a == 0) && ( b == 1 ) )
{
    doSomething();
}
else
{
    doThat();     // rare case
}
```

3.4 Declarations

3.4.1 Number per Line

One declaration per line is recommended since it encourages commenting. In other words,

```
int level = 1;  // indentation level
int size = 50;  // size of table
```

is preferred over

```
int level = 50, size = 50;
```

Both are acceptable, but be consistent throughout your code. **Do not put different types on the same line.** Example:

```
int foo, fooarray[]; //WRONG!
```

3.4.2 Initialization

Local variables should be initialized where they are declared. The only reason not to initialize a variable where it is declared is if the initial value depends on a needed computation that must occur to establish that initial value.

3.4.3 Placement

Declarations should only be put at the beginning of blocks. (A block is any code surrounded by curly braces "{"and "}"). Do not wait to declare variables until their first use; it can be confusing.

```
void myMethod()
{
    int int1 = 0;              // beginning of method block

    if (condition)
    {
        int int2 = 0;          // beginning of "if" block
        ...
    }
}
```

The one exception to the rule is indexes of for loops, which in Java can be declared in the "for statement":

```
for (int i = 0; i < maxLoops; i++) { ... }
```

Avoid local declarations that hide declarations at higher levels. For example, do not declare the same variable name in an inner block:

```
int count;
...
void myMethod()
{
    if (condition)
    {
        int count; // AVOID!
        ...
    }
    ...
}
```

3.5 Class and Interface Declarations

When coding Java classes and interfaces, the following formatting rules should be followed:

- No space between a method name and the parenthesis "(" starting its parameter list
- Open brace "{" appears at the end of the same line as the declaration statement or on the following line when using the "One True Bracing Style".
- Closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the "}" should appear immediately after the "{"

```
class Sample extends Object
{
    int ivar1 = 0;
    int ivar2 = 0;

    Sample(int i, int j)
    {
        ivar1 = i;
        ivar2 = j;
    }

    int emptyMethod()
    {
    }
    ...
}
```

- Methods are separated by at least 2 blank lines.

3.6 Statements

3.6.1 Simple Statements

Each line should contain at most one statement. Example:

```
argv++; // Correct
argc++; // Correct
argv++; argc--; // AVOID!
```

3.6.2 Compound Statements

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

- The enclosed statements should be indented one additional level than the compound statement.
- The opening brace should be on its own line at the same level of indentation as

the beginning of the compound statement (preferred Allman style) or at the end of the line that begins the compound statement (OTBS); the closing brace should begin a line and be indented to the beginning of the compound statement.

- Braces are used around all statements, even single statements, when they are part of a control structure, such as an "if-else" or "for statement". This makes it easier to add statements without accidentally introducing bugs due to forgetting to add braces.

3.6.3 Return Statements

A return statement with a value should not use parentheses unless they make the return value more obvious in some way. Example:

```
return;
return length;
return myDisk.size();
```

3.6.4 if, if-else, if else-if else Statements

All 'if' statements should use braces, whether using the preferred "Allman" or the Kernighan and Ritchie style. The following error-prone form (no braces) should be avoided:

```
if (condition)
    statement;
```

3.6.5 for Statements

An empty for statement (one in which all the work is done in the initialization, condition, and update clauses) should have the following form:

```
for (initialization; condition; update);
```

When using this form, be sure to comment what is being done in this empty for statement.

When using the comma operator in the initialization or update clause of a "for statement", avoid the complexity of using more than three variables. If needed, use separate statements before the "for loop" (for the initialization clause) or at the end of the loop (for the update clause).

3.6.6 while Statements

As with the 'if' statement, the while statement should also use braces, unless it is an empty while statement. Empty while statements should have the following form:

```
while (condition);
```

When using an empty while statement, be sure to comment what the code is doing and why an empty while statement was used.

3.6.7 do-while Statements

A do-while statement should have the following form:

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

An alternative form is:

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

3.6.8 switch Statements

A switch statement should have the following form:

```
switch (condition)
{
    case ABC:
        statements;
        /* falls through */
    case DEF:
        statements;
        break;
    case TUV:
    case XYZ:
        statements;
        break;
    default:
        statements;
        break;
}
```

Every time a case falls through (does not include a break statement), a comment should be added where the break statement would normally be. This is shown in the preceding code example with the /* falls through */ comment. Also note the case where TUV or XYZ will execute the same block of statements and that each case is on its own line. Do NOT place multiple case statements on the same line.

Every switch statement should include a default case. The break in the default case is redundant, but it prevents a fall-through error if later another case is added.

3.6.9 try-catch Statements

A try-catch statement is a good way to handle exceptions thrown when executing the code. A try-catch statement should have the following format:

```
try
{
    statements;
}
catch (ExceptionClass e)
{
    statements;
}
```

A try-catch statement may also be followed by “finally”, which executes regardless of whether or not the “try block” has completed successfully.

```
try
{
    statements;
}
catch (ExceptionClass e)
{
    statements;
}
finally
{
    statements;
}
```

3.7 White Space

3.7.1 Blank Lines

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

Two or more blank lines should always be used between methods.

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

- Between the local variables definitions in a method and its first statement.
- Before a block (see section 3.3.1) or single-line (see section 3.3.2) comment.
- Between logical sections inside a method to improve readability.
- After a comment, before the next statement.
- Before and after a control structure

3.7.2 Blank Spaces

Blank spaces should be used in the following circumstances:

- A keyword followed by a parenthesis should be separated by a space. Example:

```
while (true)
{
     ...
}
```

- A blank space should appear after commas in the argument lists.
- All binary operators except for (**.**) should be separated from their operands by spaces. Blank spaces should never separate unary operators such as unary minus, increment ("++"), and decrement ("--") from their operands.

  Example:

```
a += c + d;
a = (a + b) / (c * d);

while (d++ = s++){
    n++;
}

printSize ("size is " + foo + "\n");
```

- The expressions in a "for" statement should be separated by blank spaces. Example:

```
for (expr1; expr2; expr3)
```

3.8 Naming Conventions

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

3.8.1 Packages

The prefix of a unique package name is always written in all-lowercase ASCII letters and commonly has one of the top-level domain names like com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981. Subsequent components of the package name vary according to an organization's own internal naming conventions. Such conventions might specify that certain directory name components be division, department and project, machine, or login names.

Examples:

com.sun.eng
com.apple.quicktime.v2
edu.cmu.cs.bovik.cheese

If OHD developers use this common practice, then package names within OHD would begin with gov.nws.ohd.hseb or gov.nws.ohd.hsmb. Using this practice in OHD may be excessive. Package names beginning with ohd.hseb or ohd.hsmb is considered sufficient.

3.8.2 Classes

Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Whole words should be used and acronyms and abbreviations avoided (unless the abbreviation is much more widely used than the long form, such as URL or HTML).

Examples:

class Raster;
class ImageSprite;

3.8.3 Interfaces

Interface names should be capitalized like class names.

Examples:

interface RasterDelegate;
interface Storing;

3.8.4 Methods

Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized (also know as camelcase).

Examples:

run();
runFast();
getBackground();

3.8.5 Class Variables or Attributes

Class variables, also called attributes, are in mixed case beginning with an '_' followed by a lowercase first letter. Internal words start with capital letters. Variable names should be meaningful. The choice of a variable name should be mnemonic that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are i, j, k, l, m, and n for integers; c, d, and e for characters.

Examples:

```
String _className;
int    _windowHeight = 50;
```

3.8.6 Method Variables

Same rules as class variables and attributes apply with the exception of use '_' as the first

character of the variable name. Examples:

```
int    i = 0;
char   c = ' ';
float  windowWidth = 99999.9;
```

3.8.7 Constants

The names of variables declared class constants should be all uppercase with words separated by underscores ("_").

Examples:

static final int MIN_WIDTH = 4;
static final int MAX_WIDTH = 999;
static final int GET_THE_CPU = 1;

3.9 Programming Practices

3.9.1 Providing Access to Instance and Class Variables

Do not make any instance or class variable public without a good reason. Often, instance variables do not need to be explicitly set or gotten; often that happens as a side effect of method calls.

One example of appropriate public instance variables is the case where the class is essentially a data structure, with no behavior. In other words, when using a struct instead of a class (if Java supported struct), then it's appropriate to make the class's instance variables public.

3.9.2 Referring to Class Variables and Methods

Avoid using an object to access a class (static) variable or method. Use a class name instead. For example:

```
AClass.classMethod();        //OK
anObject.classMethod();      //AVOID!
```

3.9.3 Constants

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

3.9.4 Variable Assignments

Avoid assigning several variables to the same value in a single statement. It is hard to read. Example:

```
fooBar.fChar = barFoo.lchar = 'c'; // AVOID!
```

Do not use the "assignment" operator in a place where it can be easily confused with the "equality" operator. Example:

```
if (c++ = d++)       // AVOID! (Java disallows)
{
    ...
}
```

should be written as

```
if ((c++ = d++) != 0)
{
    ...
}
```

Do not use embedded assignments in an attempt to improve run-time performance. This is the job of the compiler. Example:

```
d = (a = b + c) + r; // AVOID!
```

should be written as

```
a = b + c;
d = a + r;
```

3.9.5 Parentheses

It is generally a good idea to use parentheses liberally in expressions involving mixed operators to avoid operator precedence problems. Even if the operator precedence seems clear to one programmer, it might not be obvious to others. It should not be assumed that other programmers know precedence equally well.

```
if (a == b && c == d)           // AVOID!
if ( (a == b) && (c == d) )     // USE
```

3.9.6 Code Commented Out

Any code which has been commented out either because it may have become obsolete or because it is being updated or rewritten should also include a comment stating why it is being commented out. It is beneficial to also include in the comment when the code may be deleted.

Example:

```
/* The following code has been commented out by John Doe on
   2/1/2007.  It has been determined the code is no longer
   needed.  If still not needed on or after 10/1/2007, remove
   this comment block.

   if ( manufacturer.equals (“Synergetics”) )
   {
       getSpecs();
       decode();
   }
*/
```

References

- *Java Conventions by Sun Microsystems (http://java.sun.com/docs/codeconv/)*
- *Java Developers Guide by Jamie Jaworski*
- *The C Programming Language, Second Edition, Brian Kernighan and Dennis Ritchie*
- *Indentation Styles (http://en.wikipedia.org/wiki/Indent_style)*
- *Just Java by Peter van der Linden*
- *Writing Robust Java Code by Scott W. Amble (http://www.ambysoft.com/downloads/javaCodingStandards.pdf)*