JAVA CODING STANDARDS FOR DEVELOPERS OF INTERNET ARCHITECTURE FRAMEWORK

PREPARED BY:
SUNIL FOTEDAR, DAN CWIEK, EDDIE ROACHE
LOCKHEED MARTIN

APRIL 23, 2004

TABLE OF CONTENTS

1. INTRODUCTION

Long after the original developer of a piece of code has moved on to other projects, an important goal during development is to ensure that the code is transitioned to another developer, or to another team of developers, so that they can continue to maintain and enhance the original code without having to invest an unreasonable effort to understand it.

Coding standards are important because they lead to greater consistency within the code which leads to code that is easier to understand, which in turn means it is easier to develop and to maintain. This reduces the overall cost of the applications that one creates.

In order to facilitate CMM level 3 compliance, following are three basic goals to write a coding standards guidelines document:

- The use of these guidelines should result in readable code and should encourage adherence.
- The resulting documentation should be easy to maintain.
- This document should be a living document in that as we discover better ways to do things, it gets reflected here.

1.1 ENVIRONMENTS

The application development lifecycle for the Internet architecture framework assumes use of the environments listed below.

Workstation Development Platform

IBM WebSphere Studio Application Developer on Windows NT/2000

Development Platform

WebSphere Application Server on Sun Solaris

Validation Platform

WebSphere Application Server on Sun Solaris

Integration Platform

WebSphere Application Server on Sun Solaris

Production Platform

WebSphere Application Server on Sun Solaris

1.2 STANDARDS

Though there is no one single java coding standard, most of the concepts and material presented in this document have been borrowed from Sun Microsystems's *Code Conventions for the Java Programming Language*[1] and Netscape's *Software Coding Standards Guide for Java*.[2]

1.2.1 Internet Application Standards Workgroup

The Usability Center has convened a multi-disciplinary workgroup that is tasked with addressing and defining standards that will guide the design and development of SSA's Internet applications. These standards are included on the Usability Center's Web site at: http://ro.ba.ssa.gov/ucweb/testing/overview.asp.

[1] http://java.sun.com/docs/codeconv/

[2] http://developer.netscape.com/docs/technote/java/codestyle.html

1.2.2 Section 508 Standards

The Internet Application Standards Workgroup (IASW) maintains a reference page listing Section 508 standards that they have accepted and recommend for implementation in selected SSA Internet applications. This information is compiled and cross-referenced from a number of sources at: http://eis.ba.ssa.gov/snapp/sect508/index.html.

1.3 RELATED TASKS

While not specifically part of development, developers also would perform the tasks identified below in the course of using the current development/deployment environment. These tasks may include configuration management, version control, unit testing, and integration testing.

1.3.1 Configuration Management

Configuration management is handled through MKS. Technical leads can provide access and directions for its use.

1.3.2 Change Management

Architects, developers, validation engineers, project managers, and project planners can add suggested enhancements and observed defects to BugZilla for evaluation and consideration by the architecture team. This process and mechanism are discussed in the document *Change Management and the Architecture Framework.*

1.3.3 Unit Testing

Developers are responsible for unit testing their own code. All code checked into MKS must be tested before a build.

1.3.4 Integration Testing

The Build Manager is responsible for integration testing prior to a release.

1.3.5 Code Reviews

While formal periodic code reviews are to be undertaken, developers are responsible for peer reviewing each other’s code in accordance with the rules and guidelines defined in this document.

2. CODING STANDARDS AND STYLE GUIDE

2.1 WHY HAVE CODE STANDARDS

Code standards are important to programmers because:

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code standards improve the readability of the software, allowing engineers to understand the code more quickly and thoroughly so make the code changes easier.

2.2 GOAL OF THIS DOCUMENT

This document should be a living document in that as we discover better ways to do things, it gets reflected here.

3. SCOPE AND APPLICATION OF THESE STANDARDS

It is very important to note that this document is divided into two distinct areas -- Rules and Guidelines. It is very important to recognize the following....

- Rules are those coding standards that are "necessary and required" coding practices that have been agreed upon by members of the Java team. Everyone is expected to follow these "rules".
- Guidelines are "suggested" coding practices that have been written to recognize the need for individuality AND for common coding practices. The purpose of the guidelines is to provide a framework upon which we can all create better code. However, the guidelines are not meant to impede engineering efforts when these guidelines are found to be in direct conflict with an individual's preference, so long as that preference is implemented consistently and is well documented. Finally, because we recognize that this opens the code upon to individual stylist coding habits, it is important that these habits are well documented and will then become the basis for all other updates within the affected files, i.e. when in someone else's code do as they do.

4. GENERAL RULES

The following rules apply to classes, interfaces, methods, variables and constants:

- Use full English descriptors that accurately describe the variables/fields/classes. For example, use name like firstName, grandTotal or HelloWorld.
- Use mixed case to make names readable, class/interface names should be capitalized, first letter of method and variable names should be lower case.
- Constants should be all capitalized, with underscore to separate the words.
- Use abbreviations sparsely, but if you do so then use them intelligently. For example, if you want to use a short name for “number”, use like “nbr”, or “num”, document which one you choose for the word “number” and use only that one.
- Avoid long names (<15 characters is a good idea).
- Avoid names that are similar or differ only in case.

Examples:

- Class/Interfaces names: HelloWorld, DataManager, Servlet
- Methods names: getFirstName, doPost
- Variables names: firstName, age
- Constants names: DATA_SOURCE

5. SOURCE CODE STRUCTURE

Each Java source file should contain only one single class or interface (public or non-public). This is easier to identify the source. Files longer than 2000 lines are cumbersome and should be avoided.

Java source files have the following ordering:

- Beginning comments
- Package and Import statements
- Class and interface declarations

5.1 BEGINNING COMMENTS

All source files should begin with a comment that lists the class name, version information, date, and copyright notice:

```
/*
 * Class Id
 *
 * Version information
 *
 * Date
 *
 * Copyright notice
 */
```

Example:

```
/*
 * $Id$
 *
 * $Revision$
 *
 * $Date$
 *
 * Copyright (c) 2002 Social Security Administration.
 * All rights reserved.
 * This software is the confidential and proprietary information of
 * Social Security Administration. You shall use it only in accordance
 * with the terms of the license agreement you entered into with
 * Social Security Administration (SSA)
 */
```

The Id, $Version$ and $Date$ will be updated by the corresponding information in Source Configuration Management Tools like MKS each time the file is checked in/out

5.2 NAMING CONVENTIONS

5.2.1 Package Names

Package names should be single lowercase words.

5.2.2 Class Names

Concrete classes should use natural descriptive names, begin with a capital, and have mixed case: *FooBarReader*

5.2.3 Member Function Names

Method ("member function") names should begin with a lowercase letter with each subsequent new word in uppercase, and subsequent letters in each word in lower case.

Methods for debug-only implementation should begin with "mortMortMort". (Alternatively, they can begin with the word "debug").

Static methods should begin with a capital letter with each subsequent new word in uppercase, and subsequent letters in each word in lower case.

```
Example:
      public class MyClass
      {
          void doSomethingNeat(int aValue);
          void debugDumpToScreen();
          static void SomeClassMethod(int aValue);
      };
```

5.2.4 Component Factory Names

A component factory is a public class that implements only static methods. These static methods are "Factory functions" or "component constructors". Factory class names should include the word "Factory". Factory method names should start with the word "Make."

Example:

```
public class WidgetFactory

{

            static Button MakeButton(int aButtonType);

            static ListBox MakeListBox();

};
```

5.2.5 Function Naming Patterns

Getters and setters should begin with "get" / "set" and return the appropriate object type.

Boolean getters should use "is" or "can" as a prefix, such as "isUndoable()" rather than "getUndoable()"

The first non-comment line of most Java source files is a package statement. The package statement defines a naming space so the same Java class name can be used without naming conflict. Every Java class should have a package statement. Conventionally, the package name reflects the organization hierarchy, like com.ibm.ejb, so in SSA's application, the package name will be start with gov.ssa.

After that, import statements should follow. For example:

```
package gov.ssa.common;

import java.util.List;

import javax.naming.InitialContext;
```

If there is more import statement, it is recommended that list the import statement in the following order:

java.	(Java standard package)
javax.	(Java extension package)
org.	(Open Source)
com.	(IBM code)
gov.ssa.	(SSA code)

In each category, list the import statements alphabetically.

Also fully qualify the package names imported so it is clear which Java class the code is using.

5.3 CLASSES AND INTERFACES

All interfaces and public classes should have JavaDoc comments describing the purpose of the class (interface), guaranteed invariants, usage instructions, and/or usage examples. Also include any reminders or disclaimers about required or desired improvements. Use HTML format, with added tags:

```
@author author-name
@version version number of class
@see string
@see URL
@see classname#methodname
```

Example:

```
/**
 * A class printing Hello World to the console.
 * For example:
 * <pre>
 *        HelloWorld hello = new HelloWorld;
 *        hello.sayHello();
 * </pre>
 *
 * @see          gov.ssa.exception.HelloWorldException
 * @see          HelloWorkd#sayHello
 * @version      $version$
 * @author       Lockheed Martin
 */
class HelloWorld extends Object {
        ...
}
```

5.4 VARIABLES AND CONSTANTS

5.4.1 *Constant, Class Variables and Instance Variables*

Use Java Doc comment to describe nature, purpose, constraints, and usage of constants, class and instances variables. Use // comment to separate constants, class variables and instance variables. Use HTML format, with added tags if needed:

- List the variables in the order of public, protected, package-level, private.
- List the variables in the order of primitive type, object reference type.
- List constants before variables.

Example:

```
// ----------------------------------define constants
/**
The jndi name of the DataSource which is
used in JDNI lookup <code>javax.naming.InitialContext </code>
if the jndi name is registered, an Object is returned, if not,
an ObjectNotFoundException is thrown.
*/
public static final String dataSourceName=”jndi:Sample”;
//-----------------------------------class variables
//-----------------------------------instant variables
/**
The current number of elements.
must be non-negative, and less than or equal to capacity.
*/
protected int count;
/**
The name of the Database table queried
*/
private String tableName;
```

5.4.2 Local Variables

Define local variables before they are used (lazy declaration). Use /* */ comment to document the usage, purpose of the variables.

5.5 DECLARATION

5.5.1 Number Per Line

One declaration per line is recommended since it encourages commenting.

```
int level; // indentation level
int size;  // size of table
```

Do not put different types of variables on the same line. Example:

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

5.5.2 Initialization

Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

5.6 METHODS

Method names should follow the same convention as mentioned before**,** *5.2*.

Use Java Doc comment to describe nature, purpose, preconditions, effects, algorithmic notes, usage instructions, reminders, etc. Use HTML format, with added tags:

Java doc comments go here:

```
@param  paramName description.

@return description of return value

@exception exceptionName description

@see string

@see URL

@see classname#methodname
```

Example:

```
/**

 * Insert element at front of the sequence

 *

 * @param element the Object to be added
```

```
  * @return none
  *
  */
public void  addFirst(Object element);
```

5.7 DOCUMENTS

Please see details on:

http://java.sun.com/j2se/javadoc/writingdoccomments/index.html

5.8 GENERAL RULES

Comments should add to the clarity of the code. The following general rules apply:

- Keep comments simple
- Write comments before write code
- Document why, not only what
- Avoid decoration, do not use banner like comments

Java programs can have three kinds of comments: C style comments, single line comments and documentation comments.

5.9 C STYLE COMMENTS

- C style comments are those found in C, which are delimited by /*...*/.
- Use C style comments to provide descriptions of files, data structures and algorithms.
- A blank line to set it apart from the rest of the code should precede a block comment.

5.10 SINGLE-LINE COMMENTS

Use single line of comments internally within a member functions to document business logic, section of code and declarations of local variables.

5.11 DOCUMENTATION COMMENTS

Documentation comments (known as "doc comments") are Java-only, and are delimited by /**...*/.

5.11.1 API Specifications Documentation

Include tags in the following order:

```
* @author       (classes and interfaces only)
* @version      (classes and interfaces only)
* @param        (methods and constructors only)
* @return       (methods only)
* @exception
* @see
* @since
* @serial
* @deprecated
```

- Multiple @author tags should be listed in chronological order, with the creator of the class listed at the top.
- Multiple @param tags should be listed in argument-declaration order. This makes it easier to visually match the list to the declaration.
- Multiple @throws tags (also known as @exception) should be listed alphabetically by the exception names.

5.12 STATEMENTS

5.12.1 General Rules

Each line should contain at most one statement.

Example:

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

5.12.2 Return Statements

A return statement with a value should not use parentheses.

Example:

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

6. GENERAL CODING GUIDELINES

Guidelines are *"suggested"* coding practices that have been written to recognize the need for individuality and for common coding practices. The purpose of the guidelines is to provide a framework upon which we can all create better code. However, the guidelines are not meant to impede engineering efforts when these guidelines are found to be in direct conflict with an individual's preference, so long as that preference is implemented consistently and is well documented.

6.1 NAMING GUIDELINES

6.1.1 GENERAL

All type declarations should follow standard java package naming.

- **Package** : All projects need to pick a prefix (we usually choose 2 letters). The project may split itself into any number of packages below this prefix. All package names are entirely lower case. Remember, java tools are case sensitive, and as of this writing somewhat inconsistent across platforms concerning package names.
- **Interface** : Everyone is encouraged to use the prefix "*I*" in your interface classes, such as I*Part.* This is optional, but strongly encouraged.
- **Constants :** begin with *k*: `kMyConstant`
- **Global Types :** begin with *g*: `gDays` (or "globalDays" is ok.)

6.1.2 MEMBER DATA NAMES

All member data will begin with a lowercase *f*, and then follow the normal naming convention for other identifiers (mixed case, each word beginning with uppercase). Keep in mind that providing publicly visible member data reflects negatively on your family. Use access functions instead. Respect the Java Beans coding patterns for member access unless you have a good reason not to.

Example:

```
public class CxMyClass {

    public void setMyOrdinalValue(int aValue);

    public int getMyOrdinalValue();

    private int fMyOrdinalValue;

};
```

6.1.3 PARAMETER NAMES

Parameter names should be constructed like identifiers, and include the type in the name if the type is not explicit. They begin with lowercase letters, and they typically start with the letter "*a*". Some may feel "*an*" is more readable for parameters that start with a vowel, such as "anObject."

Example:

```
public boolean numberIsEven(int aValue, Object anObject)  {
  ...
}
```

6.1.4 Layout of Source Files (*.java)

The layout for a class will be broken up into the following main sections: *Copyright Notice*, *File Description*; *Package Name*, *Imports, Constants*, *Methods, protected and private members.* Each section will be prefaced by an appropriate header block defining the section.

6.1.4.1 COPYRIGHT NOTICE

This will be the standard "legalese" copyright notice.

```
/* Copyright Notice =====================================*
 * This file contains proprietary information of Social
Security Administration.
 * Copying or reproduction without prior written approval is
prohibited.
 * Copyright (c) 2004 =====================================* /
```

6.1.4.2 FILE DESCRIPTION

This is a brief description of what the file is, does and represents. It should describe the overview of how to use the file (or classes therein). JavaDoc compatible commenting style is required. See the JavaDoc documentation available from JavaSoft.[3]

```
/* Description
 * Appropriate Description.
 * Description
 */
```

[3] http://java.sun.com/software/jdk/javadoc/index.html

6.1.4.3 PACKAGE NAME

Package names should occur on the first non-commented line of the source file and should following the naming conventions defined in this document.[4]

6.1.4.4 IMPORTS

Immediately following the package name should be the imported class names.

6.1.4.5 CONSTANTS

See the naming conventions defined in this document.

6.1.4.6 METHOD DECLARATIONS

Each method is preceded by a description in JavaDoc format. Public methods must have a standard javadoc comment header. These must be professionally commented. Remember, you never know what code will eventually be made public outside of Netscape.

Standard access methods may be grouped without a description (access methods assign or return an object attribute). Optionally, methods may be grouped within logical groupings. Here are some suggestions...

- Factory
- Private
- Protected
- Interfaces
- Accessor
- Temporal (fickle)
- I/O
- Debugging

Example:

[4] http://developer.netscape.com/docs/technote/java/codestyle.html#Naming Conventions#Naming Conventions

```
/*
======================================================= *
CONSTRUCTOR/DESTRUCTOR METHODS
* =======================================================
*/

    public CkSomeClass();
    public CkSomeClass(CkSomeClass aSourceToCopy);

/*
======================================================= *
FACTORY METHODS (usually static)
* =======================================================
*/

/** brief description  */

    public CkSomeClass createSomeClass(void){ }

/*
======================================================= *
ACCESSOR METHODS
* =======================================================
*/

    public int getState(void);
    public void setState(int aNewValue);

/*
======================================================= *
STANDARD METHODS
* =======================================================
*/

//...whatever these might be...

/*
======================================================= *
DEBUGGING METHODS
* =======================================================
*/

    void DoUnitTest(void);
```

6.1.4.7 CLASS DEFINITIONS

This section is the actual implementation of the class. Each method (and private function) will be prefaced by the standard documentation header. Read the Documentation for Methods and Functions defined in this document.[5]

6.1.5 Documentation for Methods and Functions

The following standard has been established for the documentation of methods and functions. Optional items are to be included only where applicable. A complete description of each section in the routine documentation follows.

The header is designed to be easy to use. Many items are optional, and may be omitted where they are not applicable. Each line item begins with an asterisk, and ends with blank space. You do not place an asterisk next to each line in a multiline component, only the topmost, next to the title. All subsequent lines in multiline component are to be indented so that they line up vertically with the previous line.

Example:

```
/**
* <Detailed description of method.>
*
* @param         <Description of each parameter>
* @return        <explain each return type>
* @exception     <explain each exception>
* @author        <your name>   <04-22-97 3:57pm>
**/
```

6.1.5.1 DESCRIPTION

Provide a detailed description. This may include:

- intent of method
- pre and post conditions
- side effects
- dependencies
- implementation notes

[5] http://developer.netscape.com/docs/technote/java/codestyle.html#Documentation for Methods and Functions#Documentation for Methods and Functions

- who should be calling this method
- whether the method should or should not be overridden
- where to invoke super when overriding
- control flow or state dependencies that need to exist before calling this method.

6.1.5.2 PARMS SECTION

Describes the type, class, or protocol of all the method or routine arguments. Should describe the parameters intended use (in, out, in/out) and constraints.

Example:

```
*@param    aSource - the input source string.  Cannot be
0-length.
```

6.1.5.3 RETURNS SECTION

This section is used to describe the method/routine return type. Specifically, it needs to detail the actual data type returned, the range of possible return values, and where applicable, error information returned by the method. Also note that if the return value is self, you do not have to specify a type (defaults to ID). All other cases require an explicit return type specification.

Example:

```
* @return   Possible values are 1..n.
```

6.1.5.4 EXCEPTION SECTION

The purpose section is a complete description of all the non-system exceptions that this method throws. A description about whether the exception is recoverable or not should be included. If applicable, a recovery strategy for the exception can be described here. Remember, when you throw a recoverable exception, you MUST reset the class state for re-entrancy!

Example:

```
        * @exception ResourceNotFoundException. recoverable, try
another resource
```

6.1.6 Java Grammar

Wherever appropriate, avoid code that embeds many operations in a single line. For example, avoid:

```
someObject.doThis(i++, otherObject.someField.doThat(), x?y:z)
```

This kind of code is error prone, difficult to decipher, and hard to maintain.

6.1.7 Parentheses

Parentheses are recommended for Boolean expressions to ensure proper evaluation .

Example:

```
if (((someValue<foo(theParm)) && anotherValue) ||
todayIsMyBirthday)
```

TIP: Place constants on the left side of your expressions; assignment, boolean and otherwise. This technique can catch assignment (versus equality) errors, as well as promote constant factoring for poor quality compilers.

6.1.8 Constants

Constants will use mixed case and begin with lowercase k. They offer compile time type checking.

Example:

```
static final float kPi = 3.14159;

static final int kDaysInWeek = 7;
```

6.1.9 Magic Numbers

Literal ordinal constants embedded within source should *almost* never be used. Whenever possible, use constants instead of literal ordinal constants:

Example:

```
int totalDays = 10 * DAYSINWEEK;        //boo, hiss!

static final int kDaysInWeek = 7;       //hooray! hooray!
```

6.1.10 Debugging

1. First and foremost, understand how to write solid code. Then go back and reread the java programming language reference documentation from JavaSoft to refresh you memory on the precedence order of operators.[6]

2. Now go read the internal document on abnormal condition handling.

[6] http://java.sun.com/software/jdk/1.1/docs/api/packages.html

3. Make sure *every* path of execution through your code has been thoroughly tested. You can never do enough coverage testing. Here's a neat idea: try actually stepping through each line! While you're hard at work testing your code, be sure to throw invalid inputs at every public interface you provide. The last thing we want to do is crash because *your* routine didn't handle invalid inputs from *ours.* Never break the build.

4. Want to know the secret to fast code? Calculate those comparison values once: reuse them often. Avoid global variables too, since they can wreak havoc with the pre-fetch instruction queue.

5. Don't forget to adhere to our Orthodox Canonical Form. This is a template that, when followed, guarantees that you have all the default methods your class needs to be a good java citizen. It is our policy to produce code that can test itself to the greatest extent possible. To that end, we encourage the use of three debugging techniques: asserts, pre/post conditions and self testing methods.

6.1.10.1 Asserts

It is highly recommended that assert methods (warnings/errors) be used to aid in debugging, especially to verify assumptions made in code. This will be the technique used for reporting run-time errors and warnings.

6.1.10.2 Pre and Post Conditions

In order to bulletproof your code, you should use asserts to test all boundary conditions. You're not doing any favors for anyone by "defensive" programming practices that allow clients of your code to make improper calls. In other words, it is better to blow up in debug builds if you are given bad data rather than trying to "fix" the data and hide bugs.

Example:

```
String copyString(String aOtherString)

{

   Assert.PreCondition(NULL != aOtherStr, "Null string 
given");

    ...

   Assert.PostCondition(fSelfString.length()
>=aOtherString.length(),

            "lengths don't match after copy.");

}
```

We do not want debug code to be included into release builds. Therefore, all assertions are removed by the compiler (the java equivalent of #defines) by flipping the debug flag in the Assert class.

6.1.10.3 SelfTest Methods

Each package will include a SelfTest class. This class should have routines to thoroughly unit test every class in the package. If appropriate, the SelfTest class may also contain methods for test integration between the classes in this package, and between the classes in this package and their dependencies in other packages.

Here is the rule for using SelfTest Methods: When you design your class, you should design its unit test. When you design your package, you should design your integration test. You are NOT done with the implementation of a class until its unit test is implemented and can be run successfully. You are NOT done with your package implementation until all unit tests are coded and run successfully, and the integration tests (if appropriate) are coded and run successfully.

6.1.11 Source Code Style Guidelines

6.1.11.1 LINE SPACING

- Line width should not ordinarily exceed 80 characters. Use your best judgment.
- Tab sizes should be set equal to 2 spaces and set to be expanded to spaces.

6.1.11.2 BRACES

- The starting brace must be on the next line aligned with the conditional. The ending brace must be on a separate line and aligned with the conditional. It is strongly recommended that all conditional constructs define a block of code for single lines of code.
- Some options are given below to accommodate the vast majority of programmer's styles.

6.1.11.3 IF,IF-ELSE,IF-ELSE IF-ELSE STATEMENTS

Place the IF keyword and conditional expression on the same line. The IF-ELSE class of statements should have the following form:

```
if (condition)
{
   statements;
}

if (condition)
{
   statements;
}
else
{
   statements;
}

if (condition)
{
   statements;
}
else if (condition)
{
   statements;
}
else
{
   statements;
}
```

6.1.11.4 WHILE Statements

The WHILE construct uses the same layout format as the IF construct. The WHILE keyword should appear on its own line, immediately followed by the conditional expression. The statement block is placed on the next line.

Examples:

```
while (expression)
{
  statement;
}
```

6.1.11.5 DO..WHILE STATEMENTS

The DO..WHILE form of the while construct should appear as shown below:

Examples:

```
do
{
  statement;
}
while (expression);
```

6.1.11.6 SWITCH STATEMENTS

The SWITCH construct uses the same layout format as the if construct. The SWITCH keyword should appear on its own line, immediately followed by its test expression. The statement block is placed on the next line. Every `switch` statement should include a default case.

Examples:

```
switch (expression)
{
  case n:
    statement;
    break;
  case x:
    statement;
    // Continue to default case
  default:              //always add the default case
    statement;
    break;
}
```

6.1.11.7 TRY/CATCH/FINALLY STATEMENTS

The try/catch construct is similar to the others. TRY keyword should appear on its own line; followed by the open brace; followed by the statement body; followed by the close brace on its own line. Any number of CATCH phrases are next consisting of the CATCH keyword and the exception expression on its own line; followed by the CATCH body; followed by the close brace on its own line. The FINALLY clause is the same as a CATCH.

Example:

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

6.1.11.8 FOR STATEMENTS

A for statement should have the following form:

```
for (initialization; condition; update)
{
   statements;
}
```

6.1.12 White Space

6.1.12.1 Blank Lines

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

Two blank lines should always be used in the following circumstances:

- Between sections of a source file
- Between class and interface definitions

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

- Between methods
- Between the local variables in a method and its first statement
- Before a block or single-line comment
- Between logical sections inside a method to improve readability

6.1.12.2 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) {
   ...
}
```

Note that a blank space should not be used between a method name and its opening parenthesis. This helps to distinguish keywords from method calls.

- A blank space should appear after commas in argument lists.
- All binary operators except `.` 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++;
}
```

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

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

- Casts should be followed by a blank space.
  Example:

```
myMethod((byte) aNum, (Object) x);
```

6.1.12.3 Indentation

Four spaces should be used as the unit of indentation. For formatting consistency, spaces are to be used in place of hard tabs.

6.1.12.4 Line Length

Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.

Note: Examples for use in documentation should have a shorter line length-generally no more than 70 characters.

6.1.12.5 Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:

- Break after a comma.
- Break before an operator.

- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.

If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

Here are some examples of breaking method calls:

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

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

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

```
longName1 = longName2 * (longName3 + longName4 - longName5)
           + 4 * longname6; // PREFER

longName1 = longName2 * (longName3 + longName4
                        - longName5) + 4 * longname6; // AVOID
```

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 conventional indentation, so instead it indents 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 horkingLongMethodName(int anArg,
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
            ...
}
```

Line wrapping for if statements should generally use the 8-space rule, since conventional (4 space) indentation makes seeing the body difficult. For example:

```
//DON'T USE THIS INDENTATION
if ((condition1 && condition2)
                || (condition3 && condition4)
                ||!(condition5 && condition6)) { //BAD WRAPS
                doSomethingAboutIt();            //MAKE THIS LINE
EASY TO MISS
}

//USE THIS INDENTATION INSTEAD
if ((condition1 && condition2)
        || (condition3 && condition4)
        ||!(condition5 && condition6)) {
            doSomethingAboutIt();
}

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

Here are three acceptable ways to format ternary expressions:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

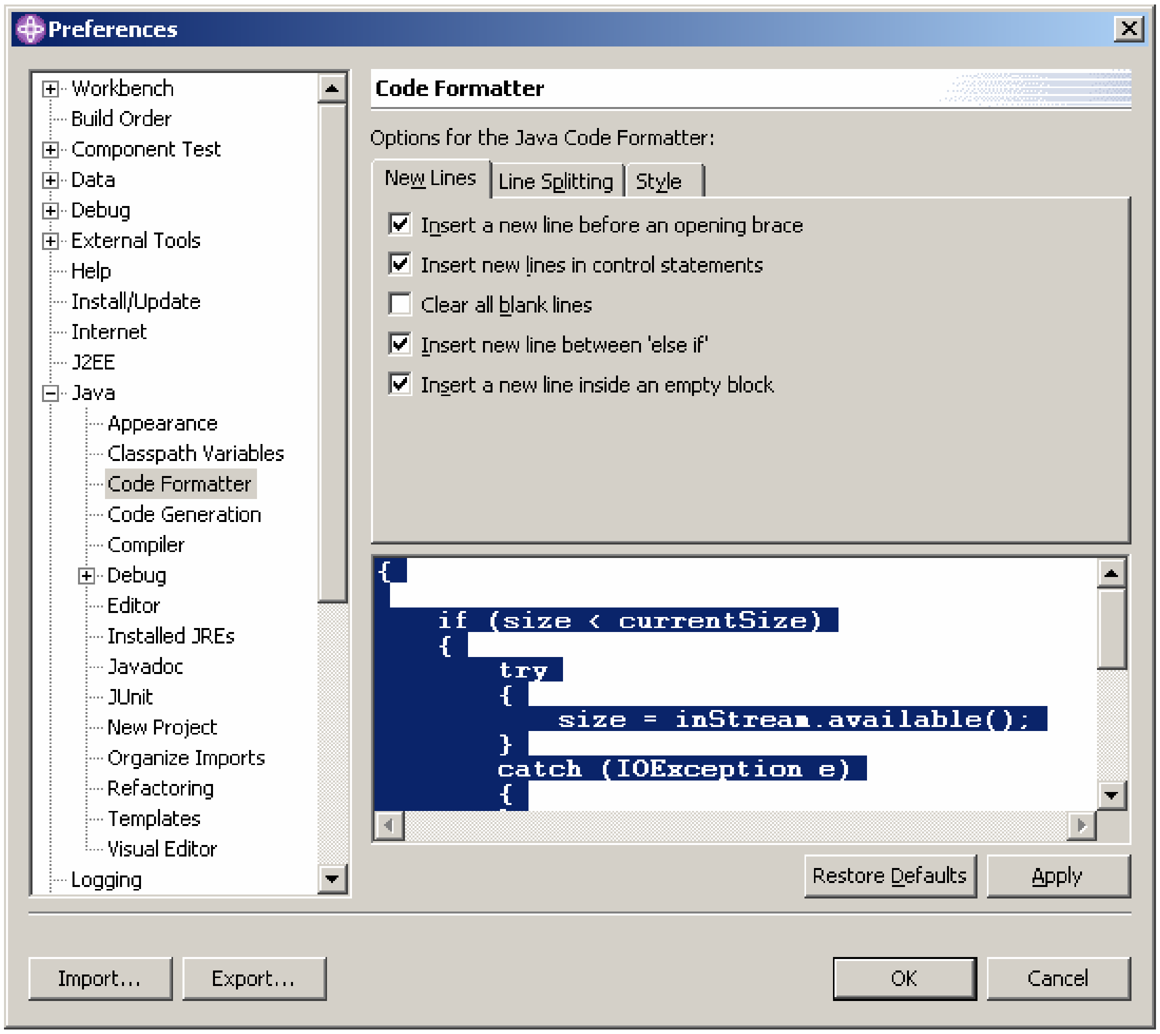
7. APPENDIX A: CODE FORMATTING

In an effort to standardize the code review process, it is recommended that all code be formatted properly.

In an effort to standardize the code review process, I am recommending that all code be formatted properly.

Within WSAD do the following:

1. Window->Preferences
2. Expand "Java"->"Code Formatting"->"New Lines". Select the following:



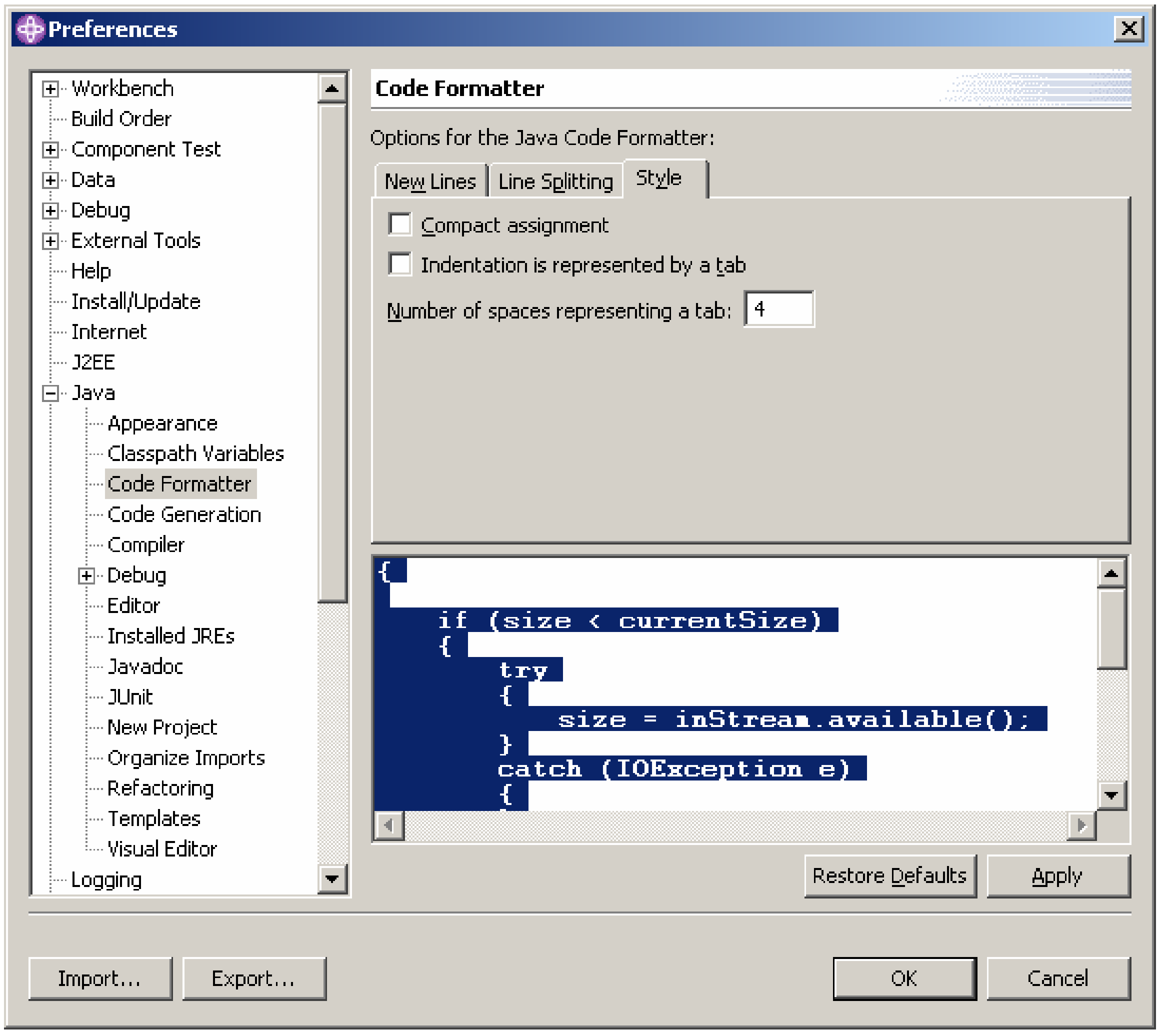
Code Formatter

Options for the Java Code Formatter:

- ✓ Insert a new line before an opening brace
- ✓ Insert new lines in control statements
- ☐ Clear all blank lines
- ✓ Insert new line between 'else if'
- ✓ Insert a new line inside an empty block

```
{
    if (size < currentSize)
    {
        try
        {
            size = inStream.available();
        }
        catch (IOException e)
        {
```

3. Go to the "Style" tab and deselect everything. Make sure "Indentation is represented by a tab" is unchecked and "Number of spaces representing a tab" is set to 4.



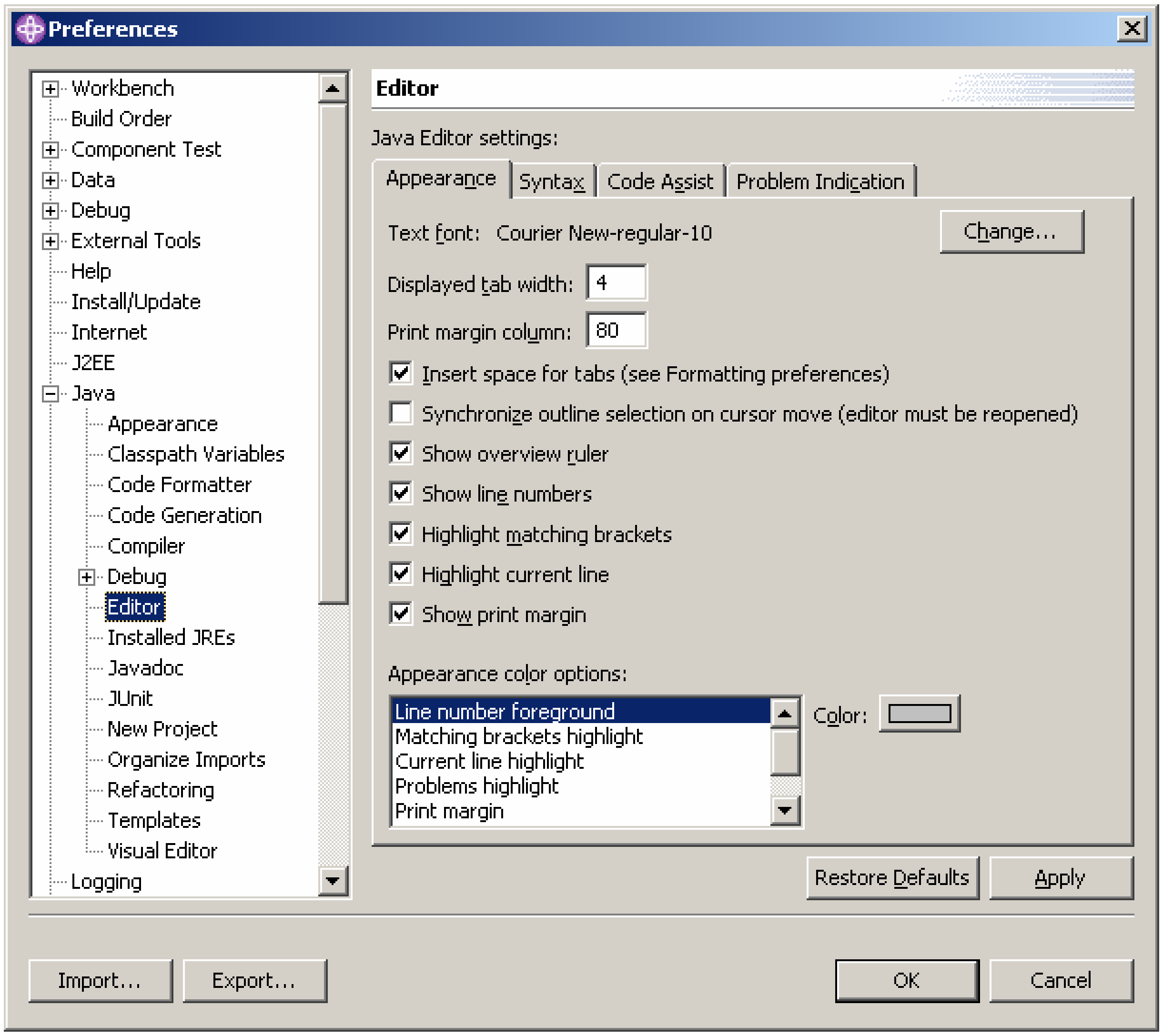
Options for the Java Code Formatter:

- [ ] Compact assignment
- [ ] Indentation is represented by a tab

Number of spaces representing a tab: 4

```
{
    if (size < currentSize)
    {
        try
        {
            size = inStream.available();
        }
        catch (IOException e)
        {
```

4. Select “Java”->”Editor”. Select “Insert spaces for tabs”



Text font: Courier New-regular-10
Displayed tab width: 4
Print margin column: 80
[✓] Insert space for tabs (see Formatting preferences)
[ ] Synchronize outline selection on cursor move (editor must be reopened)
[✓] Show overview ruler
[✓] Show line numbers
[✓] Highlight matching brackets
[✓] Highlight current line
[✓] Show print margin

5. Before you save your modified class, go to “Source”->”Format” (Hotkey Ctrl Shift F). This will reformat your class based on the parameters set above.

8. APPENDIX B: WEBSPHERE CODING STANDARDS

You can prevent many errors by following recommended WebSphere coding standards as you write code for new functionality, customization, and integration[7]. These standards help you avoid writing code that is technically valid and correct, but likely to cause functionality, performance, or scalability problems on WebSphere Application Server. The following WebSphere coding standards are among those recommended in the IBM white paper "WebSphere Application Server Development Best Practices for Performance and Scalability." http://www-1.ibm.com/support/docview.wss?uid=swg27000615

* Release HttpSessions when finished.

* Minimize synchronization in servlets.

* Do not use SingleThreadModel.

* Reuse datasources for JDBC connections.

* Release JDBC resources when done.

* Minimize use of System.out.println.

* Avoid String concatenation "+=".

* Reuse EJB homes.

* Do not use Beans.instantiate() to create new bean instances.

The complete list of standards, as well as the justification for each standard, is available at www3.ibm.com/software/webservers/appserv/ws bestpractices.pdf.

[7] http://www.findarticles.com/cf_dls/m0MLX/7_2/107140363/p2/article.jhtml?term=

9. APPENDIX C: DOCUMENTING INTERFACES WITH DESIGN BY CONTRACT

Another AEP practice, Design by Contract (DbC), prevents integration problems. Integration is an especially error-prone phase for business system projects. These projects typically involve a moderate amount of time developing code around the middleware--the heart of the system--then a great deal of time trying to ensure that the glue code, existing or purchased components, and other hardware and software interact successfully. Even if each part is solid in and of itself, problems in the interfaces between the components can cause system functionality problems--especially if the problem occurs in a critical or frequently used interface.

Design-by-contract involves formally documenting each unit's interface requirements and assumptions. When each unit's interfaces are clearly documented in a contract, developers are less likely to introduce errors by writing code that incorrectly interfaces with other units. In addition, if the code is compiled with a DbC-enabled compiler such as iContract or Parasoft's Jcontract, you receive automatic and immediate notification if interface problems arise during test executions. This allows you to perform an extensive amount of interface verification with a relatively minor investment of time and effort.

Figure 2 illustrates where contracts (represented by small squares) should be added to units (represented by circles). Ideally, contracts specify requirements that must be satisfied before the unit executes, after the unit executes, and (optionally) at various points during execution.

http://www.mmsindia.com/DBCForJava.html

Design by Contract (DBC) pioneered by Bertrand Meyer [1] is widely acknowledged to be a powerful technique for writing reliable software. The three key elements of DBC are preconditions, postconditions and class invariant. An example of a language that has direct support for DBC is Eiffel. Unfortunately, JavaTM does not directly support DBC. However, jmsassert brings the benefits of DBC to Java. In this document, you will find information that will assist you in incorporating "contract" into a Java program. jmsassert is currently available on WindowsTM platforms only.

Here is how, in a nutshell, you will use jmsassert. The java program that requires rigorous specification will be annotated at the source level by using certain markers within JavadocTM comments. The next step is to run jmsassert on the Java source code; this process results in the automatic creation of certain contract files that contain code in JMScriptTM, a Java-based scripting language developed by Man Machine Systems. The generated JMScript code actually represents triggers that are called by the assertion runtime to enforce the explicitly stated contractual obligations on the part of suppliers and consumers..

The three essential elements of DBC are precondition, postcondition and class invariant. To use jmsassert, you need to first specify these elements (any or all of them) within Javadoc comments using special tags (or markers). Table 1 lists these tags.

Tag	Description
@inv	Class invariant
@pre	Method precondtion
@post	Method postcondition

Table 1. Tags to Specify DBC Elements

Class Invariant

Class invariant is represented by the marker "@inv". The marker is followed by a boolean expression that may reference any element of the class or its direct/indirect bases, including private elements. This tag may appear within any Javadoc comment as part of a class. However, it is preferable (in the interest of readability) to define it just before the class. In the case of anonymous classes, where it is not possible to specify the invariant before the class, it may appear in any Javadoc comment as part of the class. Multiple @inv tags may appear for the same class within one or more Javadoc comments. The effective class invariant will then be the conjunction of all following boolean expressions. Here is an example of correct invariant specification.

```
/** x
   @inv a > 0
*/
class A {
   int a = 1;
   int b;
   int c;
   /**
      @inv a < 45
      @inv (b >= 100 && b < 900)
   */
   void ff() { /* ... */ }
   /**
      @inv c != b
   */
   public void gg() { /* ... */ }
}
```

The effective invariant for the above class is

```
   a > 0 && a < 45 && (b >= 100 && b < 900) && c != b
```

Precondition

The corresponding marker is "@pre". Preconditions must be defined within Javadoc comments preceding the respective methods. The marker is followed by a boolean expression as in the class invariant. The condition may reference arguments passed to the method, in addition to accessing elements of the class. If multiple @pre markers appear within the same Javadoc comment, the effective precondition for the method will be the conjunction of all such preconditions. The following is an example.

```
/**
@inv a > 0
*/

class A {
   int a = 1;
   /**
      @pre val > 100
      @pre val <= 200
   */
   void ff(int val) {
      a = val;
   }
}
```

The effective precondition for method ff() is:

```
   (val > 100 && val <= 200)
```

Postcondition

The marker used to denote a postcondition is "@post". Postconditions are specified within Javadoc comments preceding the respective methods. The condition may be any boolean expression as in the precondition case. In addition, the boolean expression may use the qualifier "$prev" on an expression to denote the expression’s value at the method’s entry point. For example, "$prev (top)" denotes the value of variable "top" at method entry (without the qualifier, it indicates its current value). The keyword "$ret" may appear in the condition to denote the method’s return value.

The following is an example.

```
class A {
   int a;
   /**
      @pre val > 10
      @post a == $prev(a) + val
   */
   public void setVal(int val) {
      a += val;
   }
   /**
      @post $ret == a
   */
   public int getVal() {
        return a;
    }
}
```

The order of tags within a Javadoc comment is not important. The following is a legal specification.

```
class A {
    /**
        @post a > 100
        @pre val > 100
        @post a <= 300
        @inv a > 0
        @post val <= 300
    */

    void ff(int val) {
        a = val;
    }
}
```

The assertion expressions that appear after tags must conform to the syntax of JMScript boolean expressions. To capture more complex conditions, one may use the special escape marker "@macro" to directly embed a JMScript code fragment. As an example of where escaping to JMScript code via @macro directive helps, consider the following class.

```
class MyVector {
    private int[] arr = new int[10];
    /**
        @post @macro foreach(elm in arr) assertPost(elm == 0);
    */
    public void clear() {
        for(int i = 0; i < 10; ++i)
            arr[i] = 0;
    }
    // Other elements
}
```

When are the contract elements checked?

It is important to know at what points in the program execution different assertions are checked. Table 2 shows how pre-, postcondition and invariant are used in the context of a class.

Event		Invariant	Pre-condition	Post-condition
Method Entry	Instance Method	Y	Y	-

	Constructor	N	Y	-
	Private Method	N	Y	-
	Static Method	N	Y	-
Method Exit	Instance Method	Y	-	Y
	Constructor	Y	-	Y
	Private Method	N	-	Y
	Static Method	N	-	Y

Table 2. JMScript Triggers Context

It can be seen from the above table that invariant is not checked in the context of private methods and postcondition is not checked when a method terminates abnormally by throwing an exception.

Recursive Calls

In case of recursive calls, JMScript executes triggers as one would expect. The triggers are executed each time the method is called irrespective of whether the call is recursive or not.

It is important to note that the trigger for a Java method is executed even if the call originates from another trigger. In particular, calling a Java method in the method's trigger can lead to a potentially recursive situation.

Contracts for Interfaces and Base Classes

Contracts may be specified for an interface, which gets propagated to its implementing classes. The same rule is applied to class hierarchies. For both interfaces and class hierarchies, preconditions are disjuncted, whereas postconditions and class invariants are conjuncted. Take the example of an interface shown below.

```
interface Employee {
   /**
      @pre age > 25
   */
   public void setAge(int age);

   /**
      @post $ret > 25
   */
   public int getAge();
}
```

Pre- and postconditions have been established for the two declared methods. Now consider a class that implements this interface:

```
class ImpEmployee implements Employee {
   private int eage;
   /**
      @pre age < 65
   */
   public void setAge(int age) {
      eage = age;
   }

   /**
      @post $ret < 65
   */
   public int getAge() {
      return eage;
   }
}
```

The effective precondition for ImpEmployee.setAge() is (age > 25 || age < 65) and the effective postcondition for ImpEmployee.getAge() is ($ret > 25 && $ret < 65). This can be extended to a class implementing multiple interfaces.

Similarly, whenever an inner class method is executed it can potentially change the outer class fields thus affecting its invariant. The test environment automatically checks the invariant of the outer class(es) when an inner class method is executed.

Table 3 summarizes how pre-, postcondition and invariant of related methods and classes are called in the context of a class.

Trigger Type	Executed Triggers	Conditions to be satisfied
Invariant	• Invariant of the class. • Invariant of all the base classes and base interfaces, if any. • Invariant of all its outer classes, if any.	• The invariant of the class, its bases classes and interfaces must all be satisfied. • The invariant of all its outer classes must be satisfied.
PreCondition	• Precondition of the method.	• At least one of the preconditions must be true

	• Precondition of all the methods that it overrides	(disjuncted).
PostCondition	• Postcondition of the method. • Postcondition of all the methods that it overrides	• All the postconditions must be true (conjuncted).

Table 3. JMScript Trigger Execution Pattern.

Illustrative Example

The following discussion serves to illustrate our approach. Consider the canonical stack implementation in Java (to avoid confusion with JDK Stack class, the following class has been named MyStack):

```
/**
   @inv (top >= 0 && top < max)
*/
class MyStack {
   private Object[] elems;
   private int top, max;

   /**
      @pre (sz > 0)
      @post (max == sz && elems != null)
   */
   public MyStack(int sz) {
      max = sz;
      elems = new Object[sz];
   }

   /**
      @pre !isFull()
      @post (top == $prev (top) + 1) && elems[top-1] == obj
   */
   public void push(Object obj) {
      elems[top++] = obj;
   }

   /**
      @pre !isEmpty()
      @post (top == $prev (top) - 1) && $ret == elems[top]
   */
   public Object pop() {
      return elems[--top];
   }

   /**
      @post ($ret == (top == max))
   */
   public boolean isFull() {
      return top == max;
   }

   /**
      @post ($ret == (top == 0))
   */
   public boolean isEmpty() {
      return top == 0;
   }
} // End MyStack
```

Once a Java source file has been annotated, as above, the next step is to use the preprocessor "jmsassert" utility to generate JMScript triggers for the various assertions. In this case, we do the following:

```
> jmsassert –s MyStack.java
```

The preprocessor creates two output files:

1. default_MyStack.jms
2. Startup.jms

The first file (see Figure 1) contains JMScript triggers corresponding to the embedded assertions in Java source. The second file (Figure 2) makes it convenient to register the automatically generated triggers (in particular, if there are more than one class) with JMScript runtime.

```
/*
 * Trigger file for class #default.MyStack.
 * Generated by JMSAssert on Monday, April 12, 1999.
 * Any changes you make to this file will be overwritten if
 * you regenerate this file.
*/

import macro;

// Postcondition for method - MyStack(int)
MyStackPost(meth, $obj, sz) {
    assertPost(($obj.max == sz && $obj.elems != null));
}

// Precondition for method - MyStack(int)
MyStackPre(meth, $obj, sz) {
    assertPre((sz > 0));
}

// Postcondition for method - void push(Object)
pushPost(meth, $obj, obj, $ret) {
    assertPost(($obj.top == this.top$prev + 1) && $obj.elems[this.top$prev] == obj);
}

// Precondition for method - void push(Object)
pushPre(meth, $obj, obj) {
    this.top$prev = $obj.top;
    assertPre(!$obj.isFull());
}

// Postcondition for method - Object pop()
popPost(meth, $obj, $ret) {
    assertPost(($obj.top == this.top$prev - 1) && $ret == $obj.elems[$obj.top]);
}

// Precondition for method - Object pop()
popPre(meth, $obj) {
    this.top$prev = $obj.top;
    assertPre(!$obj.isEmpty());
}

// Postcondition for method - boolean isFull()
isFullPost(meth, $obj, $ret) {
    assertPost(($ret == ($obj.top == $obj.max)));
}

// Postcondition for method - boolean isEmpty()
isEmptyPost(meth, $obj, $ret) {
    assertPost(($ret == ($obj.top == 0)));
}

MyStackinv(meth, $obj) {
    assertInv(($obj.top >= 0 && $obj.top <= $obj.max));
}

static {
    assertStrMyStack = {
        { "<init>(I)V", "POSTCONDITION", "MyStackPost" },
        { "<init>(I)V", "PRECONDITION", "MyStackPre" },
        { "push(Ljava/lang/Object;)V", "POSTCONDITION", "pushPost" },
        { "push(Ljava/lang/Object;)V", "PRECONDITION", "pushPre" },
        { "pop()Ljava/lang/Object;", "POSTCONDITION", "popPost" },
        { "pop()Ljava/lang/Object;", "PRECONDITION", "popPre" },
        { "isFull()Z", "POSTCONDITION", "isFullPost" },
        { "isEmpty()Z", "POSTCONDITION", "isEmptyPost" },
        { "", "INVARIANT", "MyStackinv" }
```

```
   };
   setClassTrigger("MyStack", assertStrMyStack);
}

load() {}
```

Figure 1. Triggers in JMScript for MyStack class

```
//Startup trigger file - c:\ranga\jverify\Startup.jms
import macro;
load() {}
static {
   default_MyStack.load();
}
```

Figure 2. The Startup JMScript File

The third step is to compile the relevant Java source files. Assume that in addition to the MyStack definition as above, we also have the following test driver class:

```
class StackTest {
   public static void main(String[] args) {
      MyStack s = new MyStack(2); // Can push at most two elements
      s.push(new Integer(1));
      s.push(new Integer(23));
      s.push(new Integer(0)); // Precondition violation here!
   }
}
```

This driver, along with the MyStack class, is compiled using a regular Java compiler such as "javac". The final step is, of course, to run the Java code with assertions enabled. To do this, invoke the Java interpreter as follows:

```
>java -Xdebug -Xnoagent -Djava.compiler=NONE –Xrunjmsdll:Startup StackTest
```

The CLASSPATH environment variable must be appropriately set to JDK1.2 runtime files, and additionally, must include mmsclasses.jar, which is supplied as part of the assertion environment.

The DLL jmsdll includes the JMScript interpreter. This DLL registers itself with JVM and assigns assertion triggers to the respective Java methods. When the JVM invokes a Java method, the call is intercepted by the DLL, and if a trigger (pre-, postcondition or invariant) is associated with it, the corresponding JMScript trigger method is invoked. Notice how the trigger code in JMScript is able to access private elements of a class. We consider this to be a significant advantage in terms of testing. Though, as brought out in this article, JMScript is primarily useful for bringing DBC to Java, the language has been designed for general purpose scripting, and has some very attractive features such as partial function specialization, multimethod, dynamic inheritance, and so on. Its strength derives from the underlying JVM.

One of the benefits of this approach is that the original Java source code is unmodified; what is tested with assertions is the same as the one executed normally. To run the test driver without enabling assertion code, simply run as follows:

```
>java StackTest
```

The limitation that Java source code must be available in order to specify contracts is overcome when you use the JVerify environment. That environment allows assignment of triggers to Java by inspecting compiled class files.

Conclusion

Though the Java programming language does not directly support design by contract as yet, there are several ways to add such a support from outside. The approach described in this article uses a preprocessor to map contracts embedded in Java source code to triggers in JMScript, a Java-based scripting language. The triggers are then automatically executed by an extension DLL that includes the JMScript interpreter. To run without assertions, all that is required is to run the Java program without the extension DLL. This approach ensures that the released program is identical to the one tested. In order to test a class, no special modifications are necessary (other than source annotation). Both JVerify and JMScript are currently available on WindowsTM platforms. The assertion package described in this article is available for unrestricted free public use.

10. APPENDIX D: CODE REVIEW CHECKLIST

Code Author(s):	Owner: SSA/LM Architecture Support Team
Reviewer(s):	Review Date(s):
Component:	Review Due Date
Class:	Release:

10.1 BACKGROUND KNOWLEDGE

Familiarity with some or all of this supporting information may help you to review the artifact more effectively:

- Coding Standard
- WebSphere Architecture
- Application Structure
- JavaDocs
- Release Notes/Build List

10.2 REVIEW CHECKLIST

Task	Y/N/ NA/U	Findings/Comments
I. General Questions		
1. Does the code build correctly?		
2. Is the code you are reviewing clear to you?		
II. Coding Convention Questions		
1. Does the code respect the coding standard we are using?		
2. Does the source file start with an appropriate header and copyright information?		
3. Are variable declarations properly commented?		

Task	Y/N/ NA/U	Findings/Comments
4. Are all methods and classes documented, including parameters for input/output?		
5. Are complex algorithms adequately commented?		
6. Does code that has been commented out have an explanation?		
7. Are comments used to identify missing functionality or unresolved issues in the code?		
8. Are variable names meaningful?		
9. Do variable names avoid confusion caused by the use of similar names?		
10. Have default values been adequately commented?		
11. Is code structured logically?		
12. Are spelling and grammar correct in the printed, declaration or displayed output?		
III. Error Handling Questions		
1. Are external resources and memory released in try, catch, finally block?		
2. Are all exceptions logged properly at where they are initially thrown?		
IV. Methods Questions		
1. Do computations avoid using variables having inconsistent data type?		
2. Are mixed-format (such as integer and floating point) computations avoided?		
3. Are all intermediate and result fields of sufficient length to avoid data truncation?		

Task	Y/N/ NA/U	Findings/Comments
4. Is the precedence or evaluation sequence of the Boolean expression correct?		
5. Are non-private methods parameters explicitly verified in the code?(max, min length, null value)		
6. Does the code avoid re-writing functionality that could be achieved by using an existing API?		
V. General Comments		

10.3 ADDITIONAL COMMENTS

10.4 END NOTES

11. APPENDIX E: REFERENCES

1. Netscape's Software Coding Standards Guide for Java:
   http://developer.netscape.com/docs/technote/java/codestyle.html
2. Draft Java Coding Standard:
   http://gee.cs.oswego.edu/dl/html/javaCodingStd.html
3. AmbySoft Inc. Coding Standards for java v17.01d:
   http://www.AmbySoft.com/javaCodingStandards.pdf
   http://www.ambysoft.com/javaCodingStandards.html
4. Code Conventions for the Java Programming Language:
   http://java.sun.com/docs/codeconv/
5. A Programming Style for Java: Other Style Guides:
   http://www.webcom.com/~haahr/essays/java-style/other-guides.html

11.1 RELATED DOCUMENTS

For some other standards and style guides, see

- Mark Fussell's Java Development Standards
- joodcs standards, with links to a Coding Standards Repository for various languages.
- Macadamian Technology coding conventions
- Coding Standards for C, C++, and Java by Vision 2000 CCS Package and Application Team
- AmbySoft Inc. Java Coding Standards
- Kent Sandvik's Java Coding Style Guidelines
- Solo Software Engineering home page.
- Javasoft coding standards
- Netscape coding standards
- Cedric Beust's rationale for Java coding conventions
- Jindent tool for automatic formatting.
- How To Write Unmaintainable Code by Roedy Green.

References

1. Bertrand Meyer, Object-Oriented Software Construction,
2. Reto Kramer, "iContract – The Java TM Design by Contract TM Tool", http://www.reliable-systems.com/tools

3. Andrew Duncan and Urs Holzle, "Adding Contracts to Java with Handshake", Technical Report TRCS98-32, Department of Computer Science, University of California, Santa Barbara.

4. Murat Karaorman, Urs Holzle, and John Bruno, "jContractor: A Reflective Java Library to Support Design by Contract", Technical Report TRCS98-31, Department of Computer Science, University of California, Santa Barbara.

5. Rangaraajan, K., "How Can I Test Java Classes?", Dr.Dobb's Journal , July 1999, pp 107-110.