norther.org

Tammi Application Framework

Java Coding Standard

Version 1.5

Revision History

Date	*Version*	*Description*	*Author*
02.11.02	1.0	First version	imp
15.12.02	1.1	Minor corrections	map
03.07.03	1.2	Rational link corrected	imp
25.09.03	1.3	Synced with Eclipse	imp
19.10.03	1.4	Synced with Eclipse 3.0	imp
23.03.05	1.5	Updated skin	imp

Table of Contents

1 Introduction

1.1 Purpose

This document describes the standard Java coding conventions applied in development of the Tammi Application Framework. The goal has been to write code that is easy to understand, maintain, reuse and enhance. By following a common standard, the development team creates a consistent code base, which significantly improves its productivity especially when the code base grows larger.

1.2 Scope

As code will exist after original developers have moved on to other projects, it is important to ensure that transition of code to new developers goes on smoothly. Code that is difficult to understand runs the risk of being scrapped and rewritten.

Inexperienced developers, and cowboys who do not know any better, will often fight having to follow standards. They claim they can code faster if they do it their own way. Pure hogwash. They MIGHT be able to get code out the door faster, but I doubt it. Cowboy programmers get hung up during testing when several difficult-to-find bugs crop up, and when their code needs to be enhanced it often leads to a major rewrite by them because they're the only ones who understand their code [Amby].

- 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.
- Coding standards improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create.

For the standard to work, everyone must conform to it.

1.3 Acknowledgments

This document is a modified version of the *Code Conventions for the Java™ Programming Language* [Sun]. In addition, many useful comments from *The AmbySoft Inc. Coding Standards for Java* [Amby] and *Guidelines: C++ Programming* [Rational] have been included.

1.4 References

[Amby]	Name Link	*The AmbySoft Inc. Coding Standards for Java* http://www.ambysoft.com/javaCodingStandards.html
[Javadoc]	Name Link	*Javadoc Tool Home Page* http://java.sun.com/j2se/javadoc/index.html
[Rational]	Name Link	*Guidelines: C++ Programming* http://www.upedu.org/upedu/
[Sun]	Name Link	*Code Conventions for the Java™ Programming Language* http://java.sun.com/docs/codeconv/index.html

1.5 Principles

Clear, understandable Java source code is the primary goal of most of the conventions: clear, understandable source code being a major contributing factor to software reliability and maintainability. What is meant by clear and understandable code can be captured in the following three simple fundamental principles [Rational].

1.5.1 Minimal surprise

Over its lifetime, source code is read more often than it is written, especially specifications. Ideally, code should read like an English-language description of what is being done, with the added benefit that it executes. Programs are written more for people than for computers. Reading code is a complex mental process that can be eased by uniformity, also referred to in this guide as the minimal-surprise principle. A uniform style across an entire project is a major reason for a team of software developers to agree on programming standards, and it should not be perceived as some kind of punishment or as an obstacle to creativity and productivity.

1.5.2 Single point of maintenance

Whenever possible, a design decision should be expressed at only one point in the source, and most of its consequences should be derived programmatically from this point. Violations of this principle greatly jeopardize maintainability and reliability, as well as understandability.

1.5.3 Minimal noise

Finally, as a major contribution to legibility, the minimal-noise principle is applied. That is, an effort is made to avoid cluttering the source code with visual "noise": bars, boxes, and other text with low information content or information that does not contribute to the understanding of the purpose of the software.

2 Code Organization and Style

2.1 Packages

A package in Java is primarily used for managing name spaces. Java does also allow default "friendliness" among classes inside a package. So a Java package can be thought of as a container for related classes and interfaces. The grouping of classes into packages is an important design decision, which should be considered carefully before implementation.

2.1.1 Packaging approach

Packages can be organized based on a layer pattern with application-specific business logic forming the top layer of packages, general-purpose services occupying the second layer of packages, and protocol-specific, driver-specific and other low level functionality located in bottom layer of packages.

The layer pattern provides logical partitioning of packages with certain access rules between them. The layering restricts inter-package dependencies making the resulting system to be more loosely coupled and therefore more easily maintained.

Packages within a particular layer should only depend on packages within the same layer and in the next lower layer. Failure to restrict dependencies causes architectural degradation and makes the system brittle and difficult to maintain.

Exceptions include cases where packages need direct access to lower layer services. A conscious decision should be made on how to handle primitive services needed throughout the system, such as printing, sending messages, etc.

Packages within the same layer should never contain cross-references between each other. Such dependencies indicate a flaw in the original rule, by which classes have been partitioned into packages.

Packages should not form deep hierarchies, but one layer of service packages under application-specific packages, and one layer of auxiliary packages under services, when applicable, should be sufficient.

2.1.2 Example of packaging

In Java Management Extensions (JMX™) based architecture, both application logic and general-purpose services are implemented as MBeans. In Tammi, each MBean, or a set of related MBeans, are located in their own package. Utility classes of MBeans are either in the same package or, if there is a large bunch of them, in one or more packages under the MBean package.

Utilities shared by several MBeans in different packages have been located in a separate utility package. The utility package contains one auxiliary level of packages grouping functionally related utilities into consistent sets.

Example

```
package org.norther.tammi.spray.*; // Web application layer.

package org.norther.tammi.core.*;  // General services layer.

package org.norther.tammi.acorn.*; // Common utilities layer.
```

2.2 Source files

Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you 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 ordering:

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

2.2.1 Beginning comments

All source files should begin with a comment that lists a copyright notice. Note that the comment shoud be a documentation comment strating with /** to be recognized by automatic formatting tools.

Example

```
/**
 * Copyright (c) 2004 The Norther Organization - http://www.norther.org.
 *
 * Tammi is free software; you can redistribute it and/or
 * modify it under the terms of the GNU General Public
 * License as published by the Free Software Foundation; either
 * version 2 of the License, or (at your option) any later version.
 *
 * Tammi is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
 * General Public License for more details.
 *
 * You should have received a copy of the GNU General Public
 * License along with Tammi; if not, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA
 */
```

2.2.2 Package and import statements

The first non-comment line of most Java source files is a package statement. After that, import statements can follow.

Example

```
package org.norther.tammmi.core.base;

import javax.management.MBeanServer;
```

2.2.3 Class and interface declarations

The class or interface declaration follows the import statements. The declaration should always begin with a javadoc [Javadoc] description of the purpose of the class or interface.

2.3 Code style

One way to improve the readability of a member function is to paragraph it, or in other words indent your code within the scope of a code block. Any code within braces, the { and } characters, forms a block.

Four spaces should be used as the unit of indentation. Tabs should not be used.

2.3.1 Indention of blocks

The compound or block statement delimiters, should be at the same level of indentation as surrounding statements (by implication, this means that {} are vertically aligned). Statements within the block should be indented with four spaces.

Even one line blocks should be enclosed between braces (e.g. if-else blocks).

Case labels of a switch statement should be indented by 1 indention level from the switch statement. Statements within the switch statement can then be indented by 2 indentation levels from the switch statement and by 1 indentation level from the case labels.

Examples

```
if (true)
{
    // Statement(s) within a block indented by 4 spaces.
    foo();
}
else
{
    bar();
}

while (expression)
{
    statement();
}

switch (i)
{
    case 1:
        doSomething(); // Statements indented by
        doSomething(); // 2 indentation levels from
        break;         // the switch statement itself.

    Case 2:
        //...

    default:
        //...
}
```

2.3.2 Indention of member function declarations

Place parameters of a member function on the same line as the member function name if they fit. Place each parameter of a long parameter list on a new line indented by one indention level.

Example

```
public void someMethod(
    SomeType firstParameter,
    SomeOtherType secondParameter,
    StatusType andSubsequent);
```

2.3.3 Line length

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

Notes

If the level of indentation causes deeply nested statements to drift too far to the right, and statements to extend much beyond the right margin, then it is probably a good time to consider breaking the code into smaller, more manageable, functions.

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

2.3.4 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 4 spaces instead.

Examples

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

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

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

Examples

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

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

Line wrapping for if statements should generally use the same level rule, since (4 space) indentation makes the statements difficult to read.

Examples

```
// Don't use this indention.
if ((condition1 && condition2)
        || (condition3 && condition4)
            || !(condition5 && condition6))
{

    // Bad wraps.
    doSomethingAboutIt();
}
```

```
// Use this indention instead.
if ((condition1 && condition2)
    || (condition3 && condition4)
    || !(condition5 && condition6))
{
    doSomethingAboutIt();
}
```

Below are three acceptable ways to format ternary expressions.

Examples

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

alpha = (aLongBooleanExpression)
    ? beta : gamma;

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

2.3.5 Blank spaces

A keyword followed by a parenthesis should be separated by a space.

Examples

```
while (true)
{
    doLoop();
}

synchronized (this)
{
    foo = bar;
}
```

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

Examples

```
a += c + d;

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

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

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

The expressions in a for statement should be separated by blank spaces.

Example

```
for (expr1; expr2; expr3)
{
    doLoop();
}
```

Casts should be followed by a blank space.

Examples

```
myMethod((byte) aNum, (Object) x);
myMethod((int) (cp + 5), ((int) (i + 3)) + 1);
```

3 Comments

Comments should [Rational]:

- Supplement source code by explaining what is not obvious; they should not duplicate the language syntax or semantics.
- Help the reader to grasp the background concepts, the dependencies, and especially complex data encoding or algorithms.
- Highlight deviations from coding or design standards, the use of restricted features and special “tricks”.

3.1 General recommendations

Conform to the following recommendations [Rational]:

- Place comments near the code they are commenting upon and with the same level of indentation.
- Avoid end of line comments as they become often misaligned.
- Avoid visual noise, suck as vertical bars, closed frames or boxes.
- Use blank lines to separate related blocks of source code rather than comment lines.
- Use an empty comment line, rather than an empty line, to separate comment paragraphs
- Avoid repeating program identifiers in comments, and replicating information found elsewhere - provide a pointer to the information instead.
- Write self-documenting code rather than comments by choosing better names, using extra temporary variables, or re-structuring code.
- Take care with style, syntax, and spelling in comments.
- Use natural language comments rather than telegraphic, or cryptic style.
- Document why something is being done, not just what.

3.2 Types of comments

Java has three styles of comments: documentation (doc) comments start with /** and end with */, C-style comments which start with /* and end with */, and single-line comments that start with // and go until the end of the source-code line. Below is a summary of recommended use for each type of comment [Amby].

3.2.1 Documentation comments

Use doc comments immediately before declarations of interfaces, classes, member functions, and fields to document them. Doc comments are processed by javadoc, see the next chapter, to create external documentation for a class.

Example

```
/**
 * An input stream wrapper for reading partial streams of multipart
 * MIME messages as specified in RFC 1521.
 *
 * @author Ilkka Priha
 */
```

3.2.2 C-style comments

Use C-style comments to document outlines of code that are no longer applicable, but that you want to keep just in case your users change their minds, or because you want to temporarily turn it off while debugging. Use it also to record modifications made for correcting specific bugs, or made by a programmer who is not the primary author of the class.

Example

```
/*
This code was commented out by Foo Bar on June 4, 2001 because it
was replaced by the preceding code. Delete it after two years
if it is still not applicable.
... (the source code)
*/
```

3.2.3 Single line comments

Use single line comments internally within member functions to document business logic, sections of code, and declarations of temporary variables.

Example

```
// Note that listeners can't be removed by the broadcaster,
// because the server has wrapped them with its internal one.
```

3.3 Javadoc

Javadoc [Javadoc] is a tool from Sun Microsystems for generating API documentation in HTML format from doc comments in source code. Javadoc is used for documenting classes and member functions.

3.3.1 Document classes and member functions

Although self-documenting code is preferred over comments, there is generally a need to provide information beyond an explanation of complicated parts of the code. The information that is needed is documentation of the following [Rational]:

- The purpose of each class.
- The purpose of each member function.
- The meaning of each parameter of a member function, and pre- and post-conditions on them, if any.
- The meaning of any return values; e.g., the meaning of a boolean return value for a non-predicate member function, that is, does a true value mean the member function was successful.
- Conditions under which exceptions are raised.
- Additional data accessed, especially if it is modified: important for member functions with side effects.
- Any limitations or additional information needed to properly use the class or member function.
- Any invariants or additional constraints that cannot be expressed by the language.

3.3.2 General form of a doc comment

A doc comment is made up of two parts - a description followed by zero or more tags, with a blank line (containing a single asterisk (*)) between these two sections.

Example

```
/**
 * This is the description part of a doc comment.
 *
 * @tag comment for the tag.
 */
```

The first line is indented to line up with the code below the comment, and starts with the begin-comment symbol (/**) followed by a return.

Subsequent lines start with an asterisk (*). They are indented an additional space so the asterisks line up. A space separates the asterisk from the descriptive text or tag that follows it.

Insert a blank comment line between the description and the list of tags, as shown.

The description consists of one or more capitalized sentences ending to a period. The tag comment should be a non-capitalized sentence ending to a period.

The last line begins with the end-comment symbol (*/) indented so the asterisks line up and followed by a return. Note that the end-comment symbol contains only a single asterisk (*).

Break any doc comment lines exceeding 80 characters in length, if possible. If you have more than one paragraph in the doc comment, separate the paragraphs with a <p> paragraph tag.

3.3.3 Descriptions

The first sentence of each doc comment should be a summary sentence, containing a concise but complete description of the item. This means the first sentence of each member, class, interface or package description. The javadoc tool copies this first sentence to the appropriate member, class/interface or package summary. This makes it important to write crisp and informative initial sentences that can stand on their own. This sentence ends at the first period that is followed by a blank, tab, or line terminator, or at the first tag.

3.3.4 Tag conventions

Tagged paragraphs identify certain information that has a routine structure, such as the intended purpose of each parameter of a member function, in a form that the documentation comment processor can easily marshal into standard typographical formats for purposes of presentation and cross-reference.

Different kinds of tagged paragraphs are available for class and interface declarations and for member function, field, and constructor declarations. At least the following tags should be used:

- @author, followed by the name of the author of the class (classes and interfaces only, required).
- @param, followed by the name and a short description of a member function parameter (member functions and constructors only, required for each parameter).

- @return, followed by a short description of a returned value of a member function (member functions only, required for each return value).
- @throws, followed by the name of the exception class and a short description of the circumstances that cause the exception to be thrown (member functions and constructors only, required for each exception).
- @see, followed by a cross-reference to a class, interface, member function, constructor, field, or URL.

They should be included in the above order.

Multiple @author tags should be listed in chronological order. The creator of the class should be 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 should be listed in some logical order or alphabetically.

Tag comments can be aligned vertically to start at the same level to improve readability.

Examples

```
/**
 * ContextFilterMBean loads tools available for templates into the context.
 * The tools are used from templates with implementation specific
 * references. E.g. Velocity templates refer to context tools with
 * <code>$name</code>, where name is the registration name of the tool.
 *
 * <p>Tools implementing the ContextBindingListener interface will be
 * notified when bound to the context or unbound from it.</p>
 *
 * <p>The tools are specified as properties using the following syntax:</p>
 *
 * <pre>
 *  ##
 *  # Global tools are visible to all templates for all requests.
 *  # The same instance of a global tool is shared by all clients
 *  # and the tool must be threadsafe.
 *  # global.'name' = 'class name'
 *  global.path = org.norther.tammi.spray.content.context.PathTool
 *  ##
 *  # Session tools are instantiated once for each user session, and are
 *  # stored into the session. Each session contains its own instance of
 *  # a session tool, but as the same session can be used by several
 *  # requests, the tool should be threadsafe.
 *  # session.'name' = 'class name'
 *  session.user = org.norther.tammi.spray.authenticator.context.UserTool
 *  ##
 *  # Request tools are instantiated once for each request. Each request
 *  # will get its own instance of a request tool and the tool needs not
 *  # to be threadsafe.
 *  # request.'name' = 'class name'
 *  request.page = org.norther.tammi.spray.content.context.PageTool
 * </pre>
 *
 * <p>Derived from <code>PullService</code>
 * in the Apache Jakarta Turbine project.</p>
 *
 * @author Jason van Zyl
 * @author Sean Legassick
 * @author Ilkka Priha
 */
```

```
/**
 * Callback used by the relation service when an MBean referenced
 * in a role is unregistered. The relation service will call this
 * method to let the relation take action to reflect the impact
 * of such unregistration. The user is not expected to call this method.
 *
 * @param theObjName the object name of the unregistered Mbean.
 * @param theRoleName the name of role where the MBean is referenced.
 * @throws RoleNotFoundException if there is no role with the given name.
 * @throws InvalidRoleValueException if the value provided for the role is
 *             not valid.
 * @throws RelationNotFoundException if the relation has not been added in
 *             the service.
 * @throws RelationTypeNotFoundException if the relation type has not been
 *             declared.
 * @throws RelationServiceNotRegisteredException if the service is not
 *             registered.
 */
```

4 Naming

4.1 Naming packages

There are several rules associated with the naming of packages. In order, these rules are [Amby]:

- Identifiers are separated by periods. For example, the package name java.awt is comprised of two identifiers, java and awt.
- The standard java distribution packages from Sun begin with the identifier "java" or "javax". Sun has reserved this right so that the standard java packages are named in a consistent manner regardless of the vendor of your Java development environment.
- Global package names begin with the reversed Internet domain name for your organization, with the top-level domain type in lower case. The prefix should be one of the standard Internet top-level domain names (com, edu, gov, mil, net, org, or a country specific name).
- Package names should be in singular form.

Example

```
org.norther.tammi.spray.filter;
```

4.2 Naming 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.

Use whole words - avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML). Acronyms less than four characters long are written with capitals, otherwise capitalize only the first letter.

Class names should be in singular form.

Examples

```
URLDecoder
FileStream
String
HttpRequest
```

4.3 Naming interfaces

Interface names should be capitalized like class names. The preferred Java convention for the name of an interface is to use a descriptive adjective, such as Runnable or Cloneable, although descriptive nouns, such as Singleton or DataInput, can also be used.

Note that an MBean interface always has the same name than the class implementing it suffixed with MBean.

Examples

```
Cloneable
LoggerMBean
```

4.4 Naming member functions

Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.

Examples

```
run()

runFast()
```

4.4.1 Naming accessor member functions

Getters are member functions that return the value of a field. You should prefix the word "get" to the name of the field, unless it is a boolean field and then you prefix "is" to the name of the field instead of get.

Setters, also known as mutators, are member functions that modify the values of a field. You should prefix the word "set" to the name of the field, regardless of the field type.

Examples

```
getFirstName()

setFirstName(String aName)

isAtEnd()

setAtEnd(boolean isAtEnd)
```

4.4.2 Naming constructors

Constructors are member functions that perform any necessary initialization when an object is first created. Constructors are always given the same name as their class. This naming convention is set by Sun and must be strictly adhered to.

4.5 Naming fields

You should use a full English descriptor to name your fields to make it obvious what the field represents. Fields that are collections, such as arrays or vectors, should be given names that are plural to indicate that they represent multiple values [Amby].

Example

```
firstName

zipCode

unitPrice

discountRate

orderItems
```

4.5.1 Naming constants

In Java, constants, values that do not change, are typically implemented as static final fields of classes. The recognized convention is to use full English words, all in uppercase, with underscores between the words.

The main advantage of this convention is that it helps to distinguish constants from variables.

Examples

```
MINIMUM_BALANCE
```

```
MAX_VALUE

DEFAULT_START_DATE
```

4.6 Naming variables

In general, local variables are named following the same conventions as used for fields, in other words use full English descriptors with the first letter of any non-initial word in uppercase. The names should not conflict with names with greater scope, that is, avoid name hiding [Amby].

For the sake of convenience, however, this naming convention is relaxed for several specific types of local variable:

- Streams
- Loop counters
- Exceptions

4.6.1 Naming streams

When there is a single input and/or output stream being opened, used, and then closed within a member function the common convention is to use "in" and "out" for the names of these streams, respectively. For a stream used for both input and output, the implication is to use the name "inOut".

4.6.2 Naming loop counters

Because loop counters are a very common use for local variables, and because it was acceptable in C/C++, in Java programming the use of "i", "j", or "k", is acceptable for loop counters and iterators. If you use these names for loop counters, use them consistently.

4.6.3 Naming exception objects

Because exception handling is also very common in Java coding the use of the letters "e" and "x" for a generic exception is considered acceptable.

4.6.4 Naming parameters

Parameters should be named following the exact same conventions as for local variables. As with local variables, name hiding is an issue.

Examples

```
customer

inventoryItem

photonTorpedo

in

x
```

4.7 Do not "hide" names

Name hiding refers to the practice of naming a local variable, argument, or field the same (or similar) as that of another one of greater scope. For example, if you have a field called firstName do not create a local variable or parameter called firstName, or anything close to it like firstNames or fistName. This makes your code difficult to understand and prone to bugs because other developers, or you, will misread your code while they are modifying it and make difficult to detect errors [Amby].

5 Declarations

5.1 Class declarations

Classes should be declared in a consistent manner. The common approach is to declare a class in the order of visibility:

1 public fields (accessors should be used instead)

2 protected fields (accessors should be used instead)

3 default fields (accessors should be used instead)

4 private fields

5 constructors

6 finalize()

7 public member functions

8 protected member functions

9 default member functions

10 private member functions

Static members within each grouping should be listed first, followed by instance members. Within each of these two sub-groupings, accessor member functions should be listed first as pairs of corresponding getters and setters followed by other member functions in alphabetical order.

5.2 Minimize the public and protected interface

One of the fundamentals of object-oriented design is to minimize the public interface of a class. Some of the reasons are presented below [Amby].

5.2.1 Learnability

To learn how to use a class you should only have to understand its public interface. The smaller the public interface, the easier a class is to learn.

5.2.2 Reduced coupling

Whenever the instance of one class sends a message to an instance of another class, or directly to the class itself, the two classes become coupled. Minimizing the public interface implies that you are minimizing the opportunities for coupling.

5.2.3 Greater flexibility

Whenever you want to change the way that a member function in your public interface is implemented, perhaps you want to modify what the member function returns, and then you potentially have to modify any code that invokes the member function. The smaller the public interface the greater the encapsulation and therefore the greater your flexibility.

5.3 Member function accessibility

For a good design where you minimize the coupling between classes, the general rule of thumb is to be as restrictive as possible when setting the visibility of a member function. If a member function does not have to be public then make it protected, and if it does not have to be protected then make it private [Amby].

Accessibility	***Description***	***Proper Usage***
public	A public member function can be invoked by any other member function in any other object or class.	When the member function must be accessible by objects and classes outside of the class hierarchy in which the member function is defined.
protected	A protected member function can be invoked by any member function in the class in which it is defined, any classes in the same package as that class, or any subclasses of that class.	When the member function provides behavior that is needed internally within the class hierarchy but not externally.
default	No accessibility is indicated. This is called default or package accessibility, and is sometimes referred to as friendly accessibility. The member function is effectively public to all other classes within the same package, but private to classes external to the package.	This is an interesting feature, but be careful with its use. It can be used for building domain components, collections of classes that implement a cohesive business concept such as “Customer”, to restrict access to only the classes within the component/package.
private	A private member function can only be invoked by other member functions in the class in which it is defined, but not in the subclasses.	When the member function provides behavior that is specific to the class. Private member functions are often the result of refactoring, also known as reorganizing, the behavior of other member functions within the class to encapsulate one specific behavior.

5.4 Field accessibility

When fields are declared protected there is the possibility of member functions in subclasses to directly access them, effectively increasing the coupling within a class hierarchy. This makes your classes more difficult to maintain and to enhance, therefore it should be avoided. Fields should never be accessed directly; instead accessor member functions should be used [Amby].

Accessibility	***Description***	***Proper Usage***
public	A public field can be accessed by any other member function in any other object or class.	Do not make fields public.
protected	A protected field can be accessed by any member function in the class in which it is declared, any member functions defined in classes in the same package as that class, or by any member functions defined in subclasses of that class.	Do not make fields protected.
default	A field without an access control modifier can be accessed by any member functions defined in classes in the same package as the class in which it is declared.	Do not use default accessibility.
private	A private field can only be accessed by member functions in the class in which it is declared, but not in the subclasses.	All fields should be private and be accessed by getter and setter member functions (accessors).

6 Expressions and Statements

6.1 Avoid nesting expressions too deeply

The level of nesting of an expression is defined as the number of nested sets of parentheses required to evaluate an expression from left to right if the rules of operator precedence were ignored. Too many levels of nesting make expressions harder to comprehend.

6.2 Specify the order of operations

A really easy way to improve the understandability of your code is to use parenthesis, also called "round brackets", to specify the exact order of operations in your Java code. If you have to know the order of operations for a language to understand your source code then something is seriously wrong. This is mostly an issue for logical comparisons where you AND and OR several other comparisons together. Note that if you use short, single command lines as suggested above then this really should not crop up as an issue.

6.3 Use accessor member functions

In addition to naming conventions, the maintainability of fields is achieved by the appropriate use of accessor member functions, member functions that provide the functionality to either update a field or to access its value. Accessor member functions come in two flavors: setters (also called mutators) and getters. A setter modifies the value of a variable, whereas a getter obtains it for you.

Although accessor member functions used to add overhead to your code, Java compilers are now optimized for their use, this is no longer true. Accessors help to hide the implementation details of your class. By having at most two control points from which a variable is accessed, one setter and one getter, you are able to increase the maintainability of your classes by minimizing the points at which changes need to be made.

6.4 Import classes explicitly

Avoid wildcards in import statements. Instead, import each class used by your class in a separate import statement. This practice adds a few lines to your code but makes it much easier to others (and yourself) to get an exact overview on the relationships between your code and other packages and classes.

7 Error Handling and Exceptions

The general philosophy is to use exceptions only for errors: logic and programming errors, configuration errors, corrupted data, resource exhaustion, etc. The general rule is that the systems in normal condition and in the absence of overload or hardware failure should not raise any exceptions [Rational].

7.1 Use exceptions to handle errors

Use exceptions to handle logic and programming errors, configuration errors, corrupted data, and resource exhaustion. Report exceptions by the appropriate logging mechanism as early as possible, including at the point of raise.

7.2 Minimize the number of exceptions exported from a given abstraction

In large systems, having to handle a large number of exceptions at each level makes the code difficult to read and to maintain. Sometimes the exception processing dwarfs the normal processing.

There are several ways to minimize the number of exceptions:

- Export only a few exceptions but provide "diagnosis" primitives that allow querying the faulty abstraction or the bad object for more detailed information about the nature of the problem that occurred.
- Add "exceptional" states to the objects, and provide primitives to check explicitly the validity of the objects.

7.3 Do not use exceptions for frequent, anticipated events

There are several inconveniences in using exceptions to represent conditions that are not clearly errors:

- It is confusing.
- It usually forces some disruption in the flow of control that is more difficult to understand and to maintain.
- It makes the code more painful to debug, since most source-level debuggers flag all exceptions by default.

For instance, do not use an exception as some form of extra value returned by a function (like VALUE_NOT_FOUND in a search); use a procedure with an "out" parameter, or introduce a special value meaning NOT_FOUND, or pack the returned type in a record with a discriminant NOT_FOUND.

7.4 Do not use exceptions to implement control structures

This is a special case of the previous rule: exceptions should not be used as a form of "goto" statement.

7.5 Make sure status codes have an appropriate value

When using status code returned by subprograms as an "out" parameter, always make sure a value is assigned to the "out" parameter by making this the first executable statement in the subprogram body. Systematically make all statuses a success by default or a failure by default. Think of all possible exits from the subprogram, including exception handlers.

7.6 Perform safety checks locally; do not expect your client to do so

That is, if a subprogram might produce erroneous output unless given proper input, install code in the subprogram to detect and report invalid input in a controlled manner. Do not rely on a comment that tells the client to pass proper values. It is virtually guaranteed that sooner or later that comment will be ignored, resulting in hard-to-debug errors if the invalid parameters are not detected.

8 Summary

A brief summary of the standard conventions is presented below [Amby].

8.1 Java coding conventions

Most of the time it is more important to program for people, your fellow developers, than it is to program for the computer. Making your code understandable to others is of utmost importance.

8.2 Java documentation conventions

- Comments should add to the clarity of your code.
- If your program is not worth documenting, it probably is not worth running.
- Avoid decoration, i.e. do not use banner-like comments.
- Keep comments simple.
- Write the documentation before you write the code.
- Document why something is being done, not just what.

8.3 Java naming conventions

- Use full English descriptors.
- Use terminology applicable to the domain.
- Use lower case letters in general, but capitalize the first letter of class and interface names, as well as the first letter of any non-initial word.
- Use short forms sparingly, but if you do so then use them intelligently.
- Avoid long names (less than 15 characters is a good idea).
- Avoid names that are similar or differ only in case.
- Avoid underscores.