WisDOT Java Coding Standards

Based on the AmbySoft Inc. Coding Standards for Java v17.01c

A good developer knows that there is more to development than programming.

A great developer knows that there is more to development than development.

This Version: November 8, 1999

Purpose of this White Paper

This white paper describes a collection of standards, conventions, and guidelines for writing solid Java code. They are based on sound, proven software engineering principles that lead to code that is easy to understand, to maintain, and to enhance. Furthermore, by following these coding standards your productivity as a Java developer should increase remarkably. Experience shows that by taking the time to write high- quality code right from the start, you will have a much easier time modifying it during the development process. Finally, following a common set of coding standards leads to greater consistency, making teams of developers significantly more productive.

Important Features of This White Paper

Existing standards from the industry are used wherever possible – you can reuse more than just code.

- The reasoning behind each standard is explained so that you understand why you should follow it.
- Viable alternatives, where available, are also presented along with their advantages and disadvantages so that you understand the tradeoffs that have been made.
- The standards presented in this white paper are based on real-world experience from numerous object-oriented development projects. This stuff works in practice, not just theory.
- These standards are based on proven software- engineering principles that lead to improved development productivity, greater maintainability, and greater enhancability.

Target Audience

Professional software developers who are interested in:

- Writing Java code that is easy to maintain and to enhance
- Increasing their productivity
- Working as productive members of a Java development team

Help Us Improve These Standards

Because we welcome your input and feedback, please feel free to email me at **cindy.williamson@dot.state.wi.us** with your comments and suggestions. Let's work together and learn from one another.

Acknowledgments

The team responsible for developing the initial standards draft was:

Barry Rowe
Phil Staley
Cindy Williamson

Table of Contents

1. General Concepts…… 1
1.1 Why Coding Standards are Important…… 1
1.2 The Prime Directive…… 1
1.3 What Makes up a Good Name…… 1
1.4 Good Documentation…… 2
1.4a The Three Types of Java Comments…… 3
1.4b A Quick Overview of javadoc…… 4

2. Standards for Methods…… 5
2.1 Naming Methods…… 5
2.1a Naming Accessor Methods…… 5
2.1b Getters…… 6
2.1c Setters……
2.1d Constructors……
2.2 Method Visibility…… 6
2.3 Documenting Methods…… 7
2.3a The Method Header…… 8
2.3b Internal Documentation……
2.4 Techniques for Writing Clean Code…… 9
2.4a Document Your Code…… 9
2.4b Paragraph/Indent Your Code…… 9
2.4c Use White space in Your Code…… 10
2.4d Follow the Thirty-Second Rule……
2.4e Write Short, Single Command Lines……
2.4f Specify the Order of Operations……

3. Standards for Fields (Properties)…… 11
3.1 Naming Fields…… 11
3.1a Naming Components (Widgets)…… 11
3.1b Naming Constants……
3.1c Naming Collections……
3.2 Field Visibility…… 12
3.2a Do Not "Hide" Names……
3.3 Documenting a Field…… 12
3.4 The Use of Accessor Methods…… 13
3.4a Naming Accessors…… 14
3.4b Advanced Techniques for Accessors…… 15
3.4ba Variable 16
17

Initialization…….. 18
3.4bb Getters for
Constants…… 18
3.4bc Accessors for
Collections……..
3.4bd Accessing Several Fields
Simultaneously……..
3.4c Visibility of
Accessors……..
3.4d Why Use
Accessors?……..
3.4e Why Shouldn't you Use
Accessors?……
3.5 Always Initialize Static 19
Fields……

4. Standards for Local Variables…… 20
4.1 Naming Local Variables……. 20
4.1a Naming Streams…….. 20
4.1b Naming Loop Counters……
4.1c Naming Exception Objects……..
4.2 Declaring and Documenting Local Variables…….. 20
4.2a General Comments about Declaration…….. 21

5. Standards for Parameters (Arguments) to Methods…… 22
5.1 Naming Parameters…… 22
5.2 Documenting Parameters…….. 22

6. Standards for Classes, Interfaces, Packages and Source Codes…… 23
6.1 Standards for Classes…… 23
6.1a Naming Classes…… 23
6.1b Documenting a Class…….. 24
6.1c Class Declarations…… 24
6.1d Minimize the Public and Protected Interface……
6.2 Standards for Interfaces…… 25
6.2a Naming Interfaces…… 25
6.2b Documenting Interfaces……...
6.3 Standards for Packages…… 26
6.3a Naming Packages…… 26
6.3b Documenting a Package……
6.4 Standards for Source Code Files…….. 26
6.4a Naming Source Code Files…… 26

7. Miscellaneous Standards/Issues…….. 27
7.1 Reuse……. 27
7.2 Optimizing Java Code…… 27
7.3 Writing Java Test Harnesses……. 28

8. The Secrets of Success…….. 30

8.1 Using These Standards Effectively…….. 30
8.2 Other Factors that Lead to Successful Code…….. 30

9. Summary…… 32
9.1 Java Naming Conventions……. 32
9.2 Java Documentation Conventions……. 33
9.2a Java Comment Types…… 34
9.2b What to Document……
9.3 Java Coding Conventions (General)…… 35

Glossary…… 36

References and Suggested Reading…… 40

Index…… 42

1. General Concepts

This white paper will begin with a discussion of some general concepts that are important for coding standards. It will begin with the importance of coding standards, propose the "Prime Directive" for standards, and then follow with the factors that lead to good names and good documentation. This section will set the stage for the rest of this white paper, which covers standards and guidelines for Java coding.

Throughout this white paper, there will be symbols indicating whether a particular rule is a guideline or a standard. The symbols are:

 **= guideline**

 **= standard**

1.1 - Why Coding Standards are Important

Coding standards for Java are important because they lead to greater consistency within your code and the code of your teammates. Greater consistency 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 you create.

You have to remember that your Java code will exist for a long time, long after you have moved on to other projects. An important goal during development is to ensure that you can transition your work to another developer, or to another team of developers, so that they can continue to maintain and enhance your work without having to invest an unreasonable effort to understand your code. Code that is difficult to understand runs the risk of being scrapped and rewritten – would you be proud of the fact that your code needed to be rewritten? If everyone is doing their own thing then it makes it very difficult to share code between developers, raising the cost of development and maintenance.

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. This is 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. Because they're the only ones who understand their code, they are the only ones who can do the rewrite. That is not an efficient way to operate.

1.2 - The Prime Directive

No standard is perfect and no standard is applicable to all situations: sometimes you find yourself in a situation where one or more standards do not apply. Therefore, this is what's considered to be the prime directive of standards:

When you go against a standard, document it. All standards, except for this one, can be broken. If you do so, you must document why you broke the standard, the potential implications of breaking the standard, and any conditions that may/must occur before the standard can be applied to this situation.

The bottom line is that you need to understand each standard and understand when to apply them.

1.3  What Makes Up a Good Name

Naming conventions will be discussed throughout the standards, so let's set the stage with a few basics:

1. **Use full English descriptors that accurately describe the variable/field/class/etc.** For example, use names like **firstName**, **grandTotal**, or **CorporateCustomer**. Although names like **x1**, **y1**, or **fn** are easy to type because they're short, they do not provide any indication of what they represent and result in code that is difficult to understand, maintain, and enhance (Nagler,

1995; Ambler, 1998a).

2. **Use terminology applicable to the domain.** If your users refer to their clients as "customers," then use the term **Customer** for the class, not **Client**. Many developers will make the mistake of creating generic terms for concepts when perfectly good terms already exist in that industry or domain.

3. **Use mixed case to make names readable.** You should use lower case letters in general, but capitalize the first letter of class names and interface names, as well as the first letter of any non- initial word (Kanerva, 1997).

4. **Use abbreviations sparingly, but if you do so then use them intelligently.** This means you should maintain a list of standard short forms (abbreviations), choose them wisely and use them consistently. For example, if you want to use a short form for the word "number," then choose **nbr**, **no**, or **num**, document which one you chose, and use only that one.

5. **Avoid long names (<15 characters is a good idea).** Although the class name **PhysicalOrVirtualProductOrService** might seem to be a good class name at the time, this name is simply too long and you should consider renaming it to something shorter, perhaps something like **Offering** (NPS, 1996).

6. **Avoid names that are similar or differ only in case.** For example, the variable names **persistentObject** and **persistentObjects** should not be used together, nor should **anSqlDatabase** and **anSQLDatabase** (NPS, 1996).

7. **Avoid leading or trailing underscores.** Names with leading or trailing underscores are usually reserved for system purposes, and may not be used for any user- created names except for pre- processor defines (NPS, 1996). Also, many consider underscores to be annoying and difficult to type.

1.4  Good Documentation

Documentation conventions will also be discussed, so here are some of the basics:

1. **If your program isn't worth documenting, it probably isn't worth running (Nagler, 1995).**

2. **Comments should add to the clarity of your code.** The reason why you document your code is to make it more understandable to you, your coworkers and to any other developer who comes after you (Nagler, 1995).

3. **Avoid decoration, i.e. do not use banner- like comments.** In the 1960s and 1970s COBOL programmers got into the habit of drawing boxes, typically with asterisks, around their internal comments (NPS, 1996). Sure, it seemed more artistic, but in reality it added little value to the end product. You want to write clean code, not pretty code. Furthermore, because some of the fonts used to display and print your code are proportional and others aren't, boxes may not be lined up properly anyway.

4. **Keep comments simple.** Some of the best comments are simple, point-form notes. You do not have to write a book; you just have to provide enough information so that others can understand your code.

5. **Write the documentation before you write the code.** The best way to document code is to write the comments before you write the code. This gives you an opportunity to think about how the code will work before you write it and will ensure that the documentation gets written. Alternatively, you should at least document your code as you write it. Take advantage of the fact that documentation makes your code easier to understand while you are developing it. If you are going to invest the time writing documentation you should at least get something out of it (Ambler, 1998a).

6. **Document why something is being done, not just what.** Fundamentally, most developers can figure out what a piece of code does. Look at the code in Example 1.1 below. You should see that a 5% discount is being given on orders of $1,000 dollars or more. Why is this being done? Is there a business rule that says that large orders get a discount? Is there a limited- time special on large orders or is it a permanent program? Was the original programmer just being generous? This is unknown unless it is documented somewhere, either in the source code itself or in an external document (Ambler, 1998a).

Example 1.1

```
if ( grandTotal >= 1000.00)
```

```
{
        grandTotal = grandTotal * 0.95;
}
```

1.4a The Three Types of Java Comments

Java has three styles of comments: Documentation 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. In the chart below is a summary of the standard use for each type of comment, as well as several examples.

Comment Type	Usage	Example
Documentation	Use documentation comments immediately before declarations of interfaces, classes, methods, and fields to document them. Documentation comments are processed by javadoc, see below, to create external documentation for a class.	/** Customer – A customer is any person or organization that we sell services and products to. @author J. Q. Developer */
C style	Use C- style comments to document out lines 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.	/* This code was commented out by J. T. Kirk on Dec 9, 1997 because it was replaced by the preceding code. Delete it after two years if it is still not applicable. . . . (the source code) */

Single line	Use single line comments internally within methods to document business logic, sections of code, and declarations of temporary variables.	// Apply a 5% discount to all invoices
		// over $1000 as defined by the Sarek
		// generosity campaign started in
		// Feb. of 1995.

Beware Endline Comments

McConnell (1993) argues strongly against the use of inline comments, also known as endline comments or end of line comments. He points out that the comments have to be aligned to the right of the code so that they do not interfere with the visual structure of the code. As a result they tend to be hard to format, and that "if you use many of them, it takes time to align them. Such time is not spent learning more about the code; it is dedicated solely to the tedious task of pressing the spacebar or the tab key." He also points out that endline comments are also hard to maintain because when the code on the line grows it bumps the endline comment out, and that if you are aligning them you have to do the same for the rest of them.

Therefore, endline comments should be avoided. If they are necessary, do not bother aligning them.

1.4b - A Quick Overview of javadoc

Included in Sun's Java Development Kit (JDK) is a program called *javadoc* that processes Java code files and produces external documentation, in the form of HTML files, for your Java programs. *Javadoc* is a great utility, but at the time of this writing it does have its limitations. First, it supports a limited number of tags, reserved words that mark the beginning of a documentation section. The existing tags are a very good start but are not sufficient for adequately documenting your code. This statement will be more thoroughly explained later. For now, here is a brief overview of the current *javadoc* tags in the chart below. Refer to the JDK *javadoc* documentation for further details.

Tag	Used for	Purpose
@author name	Classes, Interfaces	Indicates the author(s) of a given piece of code. One tag per author should be used..
@deprecated	Classes, Methods	Indicates that the API for the class… has been deprecated and therefore should not be used any more.
@exception name description	Methods	Describes the exceptions that a method throws. You should use one tag per exception and give the full class name for the exception.
@param name description	Methods	Used to describe a parameter passed to a method, including its type/ class and its usage. Use one tag per parameter.
@return description	Methods	Describes the return value, if any, of a method. You should indicate the type/ class and the potential use(s) of the return value.
@since	Classes, Methods	Indicates how long the item has existed, i. e. since JDK 1.1
@see ClassName	Classes, Interfaces, Methods, Fields	Generates a hypertext link in the documentation to the specified class. You can, and probably should, use a fully qualified class name.
@see ClassName# methodName	Classes, Interfaces, Methods, Fields	Generates a hypertext link in the documentation to the specified method. You can, and probably should, use a fully qualified class name.
@version text	Classes, Interfaces	Indicates the version information for a given piece of code.

The way that you document your code has a huge impact both on your own productivity and on the productivity of everyone else who later maintains and enhances it. By documenting your code early in the development process you become more productive because it forces you to think through your logic before you commit it to code. Furthermore, when you revisit code that you wrote days or weeks earlier, you can easily determine what you were thinking when you wrote it – it is documented for you already.

2. Standards for Methods

Systems professionals should always strive to maximize their productivity. Because an application spends the majority of its existence being maintained, not developed, anything that can make code easier to maintain and to enhance as well as to develop is

helpful. Never forget that the code that you write today may still be in use many years from now and will likely be maintained and enhanced by somebody other than you. You must strive to make your code as "clean" and understandable as possible, because these factors make it easier to maintain and to enhance.

In this section we will concentrate on four topics:
- Naming conventions
- Visibility
- Documentation conventions
- Techniques for writing clean Java code

2.1  Naming Methods

Methods should be named using a full English description, using mixed case with the first letter of any non- initial word capitalized. It is also common practice for the first word of a method name to be a strong, active verb.

Examples:

openAccount()
printMailingLabel()
save()
delete()

This convention results in methods whose purpose can often be determined just by looking at its name. Although this convention results in a little extra typing by the developer, because it often results in longer names, this is more than made up for by the increased understandability of your code.

2.1a - Naming Accessor Methods

Accessors, methods that get and set the values of fields (fields/ properties), along with advanced techniques for collections will be discussed in greater detail in chapter 3. The naming conventions for accessors, however, are summarized below.

2.1b  Getters

Getters are methods 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. '

Examples:

getFirstName()
getAccountNumber()
getLostEh()
isPersistent()
isAtEnd()

By following this naming convention you make it obvious that a method returns a field of an object, and for boolean getters you make it obvious that it returns true or false. Another advantage of this standard is that it follows the naming conventions used by the beans development kit (BDK) for getter methods (DeSoto, 1997).

2.1c  Setters

Setters, also known as mutators, are methods 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:

setFirstName(String aName)
setAccountNumber(int anAccountNumber)
setReasonableGoals(Vector newGoals)
setPersistent(boolean isPersistent)
setAtEnd(boolean isAtEnd)

Following this naming convention, you make it obvious that a method sets the value of a field of an object. Another advantage of this standard is that it follows the naming conventions used by the beans development kit (BDK) for setter methods (DeSoto, 1997). The main disadvantage is that ‘set’ is superfluous, requiring extra typing.

2.1d  Constructors

Constructors are methods that perform any necessary initialization when an object is first created. Constructors are always given the same name as their class. For example, a constructor for the class **Customer** would be **Customer()**. Note that the same case is used.

Examples:

Customer()
SavingsAccount()
PersistenceBroker()

This naming convention is set by Sun and must be strictly followed.

2.2  Method Visibility

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 method. In general, make methods private and only make it protected or public if absolutely necessary.

Visibility	Description	Proper Usage
public	A public method can be invoked by any other method in any other object or class.	When the method must be accessible by objects and classes outside of the class hierarchy in which the method is defined.
protected	A protected method is visible to all classes in the package and can be invoked by any method in the class in which it is defined or any subclasses of that class.	When the method provides behavior that is needed internally within the class hierarchy but not externally.
private	A private method can only be invoked by other methods in the class in which it is defined, but not in the subclasses.	When the method provides behavior that is specific to the class. Private methods are often the result of refactoring, also known as reorganizing, the behavior of other methods within the class to encapsulate one specific behavior.

2.3 - Documenting Methods

The manner in which you document a method will often be the deciding factor as to whether or not it is understandable, and therefore maintainable and extensible.

2.3a  The Method Header

Every Java method should include some sort of header, called method documentation, at the top of the source code that documents all of the information that is critical to understanding it. This information includes, but is not limited to the following:

1. **What and why the method does what it does**. By documenting what a method does you make it easier for others to determine if they can reuse your code. Documenting why it does something makes it easier for others to put your code into context. You also make it easier for others to determine whether or not a new change should actually be made to a piece of code (perhaps the reason for the new change conflicts with the reason why the code was written in the first place).

2. **What a method must be passed as parameters**. You also need to indicate what parameters, if any, must be passed to a method and how they will be used. This information is needed so that other programmers know what information to pass to a method. The *javadoc @param* tag, discussed in section 1.4b, is used for this.

3. **What a method returns**. You need to document what, if anything, a method returns so that other programmers can use the return value/ object appropriately. The *javadoc @return* tag, discussed in section 1.4b, is used for this.

4. **Known bugs.** Any outstanding problems with a method should be documented so that other developers understand the weaknesses/ difficulties with the method. If a given bug is applicable to more than one method within a class, then it should be documented for the class instead.

5. **Any exceptions that a method throws.** You should document any and all exceptions that a method throws so that other programmers know what their code will need to catch. The *javadoc @exception* tag, discussed in section 1.4b, is used for this.

6. **Visibility decisions.** If you feel that your choice of visibility for a method will be questioned by other developers, perhaps you've made a method public even though no other objects invoke the method yet, then you should document your decision. This will help to make your thinking clear to other developers so that they do not waste time worrying about why you did something questionable.

7. **How a method changes the object**. If a method changes an object, for example the **withdraw()** method of a bank account modifies the account balance, then this needs to be indicated. This information is needed so that other Java programmers know exactly how a member function invocation will affect the target object.

8. **Include a history of any code changes**. Whenever a change is made to a method you should document when the change was made, who made it, why it was made, who requested the change, who tested the change, and when it was tested and approved to be put into production. This history information is critical for the future maintenance programmers who are responsible for modifying and enhancing the code.

9. **Examples of how to invoke the method if appropriate**. One of the easiest ways to determine how a piece of code works is to look at an example. Consider including an example or two of how to invoke a method.

10. **Applicable preconditions and postconditions**. A precondition is a constraint under which a member function will function properly, and a postcondition is a property or assertion that will be true after a method is finished running (Meyer, 1988). In many ways preconditions and postconditions describe the assumptions that you have made when writing a method (Ambler, 1998a), defining exactly the boundaries of how a method is used.

11. **All concurrency issues.** Concurrency is a new and complex concept for many developers and is an old and complex topic for experienced concurrent programmers. The end result is that if you use the concurrent programming features of Java then you need to document it thoroughly. Lea (1997) suggests that when a class includes both synchronized and unsynchronized methods you must document the execution context that a method relies on, especially when it requires unrestricted access so that other developers can use your methods safely. When a setter, a method that updates a field, of a class that implements the **Runnable**

interface is not synchronized then you should document your reason(s) why. Finally, if you override or overload a method and change its synchronization you should also document why.

The important thing is that you should document something only when it adds to the clarity of your code. You wouldn't document all of the factors described above for each and every method because not all factors are applicable to every method. You would however document several of them for each method that you write.

2.3b  Internal Documentation

In addition to the method documentation, you also need to include comments within your member functions to describe your work. The goal is to make your method easier to understand, to maintain, and to enhance.

There are two types of comments that you should use to document the internals of your code: C- style comments (/* … */) and single- line comments (//). As discussed in section 1.4a, you should seriously consider choosing one style of comments for documenting the business logic of your code and one for commenting out unneeded code. The suggestion is to use single- line comments for your business logic, because you can use this style of comments both for full comment lines and for inline comments that follow at the end of a line of code. You should only use C- style comments to document out lines of unneeded code because you can easily take out several lines with only one comment. Furthermore, because C- style comments look so much like documentation comments, their use can be confusing, taking away from the understandability of the code. Therefore I use them sparingly.

Internally, you should always document:

1. **Control structures.** Describe what each control structure, such as comparison statements and loops. You shouldn't have to read all the code in a control structure to determine what it does, instead you should just have to look at a one or two line comment immediately preceding it.

2. **Why, as well as what, the code does.** You can always look at a piece of code and figure out what it does, but for code that isn't obvious you can rarely determine why it is done that way. For example, you can look at a line of code and easily determine that a 5% discount is being applied to the total of an order. That is easy. What isn't easy is figuring out WHY that discount is being applied. Obviously there is some sort of business rule that says to apply the discount, so that business rule should at least be referred to in your code so that other developers can understand why your code does what it does.

3. **Local variables.** Although we will discuss this in greater detail in chapter 4, each local variable defined in a method should be declared on its own line of code and should usually have an inline comment describing its use.

4. **Difficult or complex code.** If you find that you either can't rewrite it, or do not have the time, then you must document thoroughly any complex code in a method. The general rule of thumb is that if your code isn't obvious, then you need to document it.

5. **The processing order.** If there are statements in your code that must be executed in a defined order then you should ensure that this fact gets documented (Ambler, 1998a). There's nothing worse than making a simple modification to a piece of code only to find that it no longer works, then spending hours looking for the problem only to find that you've gotten things out of order.

Tip  **Document Your Closing Braces**

Every so often you will find that you have control structures within control structures within control structures. Although you should avoid writing code like this, sometimes you find that it is better to write it this way. The problem is that it becomes confusing which ending brace, the } character, belongs to which control structure.

The good news is that some code editors support a feature that when you select a open brace it will automatically highlight the corresponding closing one; the bad news is that not every editor supports this. Marking the ending braces with an inline comment such as **//end if**, **//end for**, **//end switch**, etc. makes the code easier to understand.

Given the choice, however, the use a more sophisticated editor may be warranted.

2.4 - Techniques for Writing Clean Code

In this section we will cover several techniques that help to separate the professional developers from the hack coders. These techniques are:

- Document your code
- Paragraph your code
- Use white space
- Follow the thirty- second rule
- Specify the order of message sends
- Write short, single command lines

2.4a - Document Your Code

Remember, if your code isn't worth documenting then it isn't worth keeping (Nagler, 1995). When you apply the documentation standards and guidelines proposed in this paper appropriately you can greatly enhance the quality of your code.

2.4b  Paragraph/ Indent Your Code

One way to improve the readability of a method 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. The basic idea is that the code within a block should be uniformly indented one unit. To ensure consistency, start your method and class declarations in column 1 (NPS, 1996).

The Java convention appears to be that the open brace is to be put on the line following the owner of the block and that the closing brace should be indented one level. The important thing as pointed out by Laffra (1997) is that your organization chooses an indentation style and sticks to it. Generally, you should use the same indentation style that your Java development environment uses for the code that it generates.

2.4c  Use White Space in Your Code

A few blank lines, called white space, added to your Java code can help to make it much more readable by breaking it up into small, easy- to- digest sections (NPS, 1996; Ambler, 1998a). The Vision 2000 team (1996) suggests using a single blank line to separate logical groups of code, such as control structures, with two blank lines to separate method definitions. Without white space it is very difficult to read and to understand.

2.4d  Follow The Thirty-Second Rule

In most situations, another programmer should be able to look at your method and be able to fully understand what it does, why it does it, and how it does it in less than 30 seconds. If he or she can't then your code is too difficult to maintain and should be improved. Thirty seconds, that's it. A good rule of thumb suggested by Stephan Marceau is that if a method is more than a screen then it is probably too long.

2.4e  Write Short, Single Command Lines

Your code should do one thing per line (Vision, 1996; Ambler, 1998a). Back in the days of punch cards, it made sense to try to get as much functionality as possible on a single line of code. Considering it has been many years since most people have even seen a punch card, this approach to writing code no longer makes sense. Whenever you attempt to do more than one thing on a single line of code you make it harder to understand. Why do this? The code should be easier to understand so that it is easier to maintain and enhance. Just like a method should do one thing and one thing only, you should only do one thing on a single line of code.

Furthermore, you should write code that remains visible on the screen (Vision, 1996). A general rule of thumb is that you shouldn't have to scroll your editing window to the right to read the entire line of code, including code that uses inline comments.

2.4f  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 (Nagler, 1995; Ambler, 1998a). If a person has 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 use AND and OR several other comparisons together. Note that if you use short, single command lines as suggested above then this really shouldn't crop up as an issue.

3. Standards for Fields (Fields/ Properties)

A field is a piece of data that describes an object or class. Fields may be base data type like a string or a float, or may be an object such as a customer or bank account. Throughout this white paper, what the Beans Development Kit (BDK) calls a property (DeSoto, 1997) will be referred to as a field.

3.1  Naming Fields

You should use a full English descriptor to name your fields (Gosling, Joy, Steele, 1996; Ambler 1997) 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.

Examples:

firstName
zipCode
unitPrice
discountRate
orderItems

3.1a  Naming Components (Widgets)

For names of components (interface widgets) you should use a full English descriptor postfixed by the widget type. This makes it easy for you to identify the purpose of the component as well as its type, making it easier to find each component in a list (many visual programming environments provide lists of all components in an applet or application and it can be confusing when everything is named **button1**, **button2**, etc.).

Examples:

okButton
customerList
fileMenu
newFileMenuItem

3.1b  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 (Gosling, Joy, Steele, 1996; Sandvik, 1996; NPS, 1996).

Examples:

MINIMUM_ BALANCE
MAX_ VALUE
DEFAULT_ START_ DATE

The main advantage of this convention is that it helps to distinguish constants from variables. We will see later in the document that you can greatly increase the flexibility and maintainability of your code by not defining constants. Instead, you should define getter methods that return the value of constants.

3.1c  Naming Collections

A collection, such as an array or a vector, should be given a pluralized name representing the types of objects stored by the array. The name should be a full English descriptor with the first letter of all non-initial words capitalized.

Examples:

customers
orderItems
aliases

The main advantage of this convention is that it helps to distinguish fields that represent multiple values (collections) from those that represent single values (non- collections).

3.2  Field Visibility

The Vision team (1996) suggests that fields should not be declared *public* for reasons of encapsulation, but this white paper will go further to state that all fields should be declared *private*. When fields are declared *protected* there is the possibility of methods 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 methods (see below) should be used.

Visibility	Description	Proper Usage
public	A public field can be accessed by any other method in any other object or class.	Do not make fields public.
protected	A protected field can be accessed by any method in the class in which it is declared or by any methods defined in subclasses of that class.	Do not make fields protected.
private	A private field can only be accessed by methods 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 methods (accessors).

For fields that are not persistent (they will not be saved to permanent storage) you should mark them as either *static* or *transient* (DeSoto, 1997). This makes them conform to the conventions of the BDK.

3.2a  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**. Try to avoid this as it 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.

3.3  Documenting a Field

Every field should be documented well enough so that other developers can understand it. To be effective, you need to document:

1. **Its description.** You need to describe a field so that people know how to use it.

2. **Document all applicable invariants.** Invariants of a field are the conditions that are always true about it. For example, an invariant about the field **dayOfMonth** might be that its value is between 1 and 31 (obviously you could get far more complex with this invariant, restricting the value of the field based on the month and the year). By documenting the restrictions on the value of a field you help to define important business rules, making it easier to understand how your code should work.

3. **Examples.** For fields that have complex business rules associated with them you should provide several example values so as to make them easier to understand. An example is often like a picture: it is worth a thousand words.

4. **Concurrency issues.** Concurrency is a new and complex concept for many developers, actually, at best it is an old and complex topic for experienced concurrent programmers. The end result is that if you use the concurrent programming features of Java then you need to document it thoroughly.

5. **Visibility decisions.** If you've declared a field to be anything but private then you should document why you have done so. Field visibility is discussed in section 3.2 above, and the use of accessor methods to support encapsulation is covered in section 3.4 below. The bottom line is that you would better have a really good reason for not declaring a variable as private.

3.4 - The Use of Accessor Methods

In addition to naming conventions, the maintainability of fields is achieved by the appropriate use of *accessor methods*, methods that provide the functionality to either update a field or to access its value. Accessor methods 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 methods used to add overhead to your code, Java compilers are now optimized for their use. 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. Optimization of Java code is discussed in section 7.2.

One of the most important standards that your organization can enforce is the use of accessors. Some developers do not want to use accessor methods because they do not want to type the few extra keystrokes required (for example, for a getter you need to type in 'get' and '()' above and beyond the name of the field). The bottom line is that the increased maintainability and extensibility from using accessors more than justifies their use.

> **Tip – Accessors Are The Only Place To Access Fields**
>
> A key concept with the appropriate use of accessor methods is that the ONLY methods that are allowed to directly work with a field are the accessor methods themselves. Yes, it is possible to directly access a private field within the methods of the class in which the field is defined but you do not want to do so. That would increase the coupling within your class.

3.4a  Naming Accessors

Getter methods should be given the name 'get' + field name, unless the field represents a boolean (true or false) and then the getter is given the name 'is' + field name. Setter methods should be given the name 'set' + field name, regardless of the field type (Gosling, Joy & Steele, 1996; DeSoto, 1997). Note that the field name is always in mixed case with the first letter of all words capitalized. This naming convention is used consistently within the JDK and is what is required for beans development.

Examples:

Field	Type	Getter name	Setter name
firstName	string	**getFirstName()**	**setFirstName()**
address	**SurfaceAddress** object	**getAddress()**	**setAddress()**
persistent	boolean	**isPersistent()**	**setPersistent()**
customerNumber	int	**getCustomerNumber()**	**setCustomerNumber()**
orderItems	Array of **OrderItem** objects	**getOrderItems()**	**setOrderItems()**

3.4b - Advanced Techniques for Accessors

Accessors can be used for more than just getting and setting the values of instance fields. In this section we will discuss how to increase the flexibility of your code by using accessors to:

- Initialize the values of fields
- Access constant values
- Access collections
- Access several fields simultaneously

3.4ba Variable Initialization

Variables need to be initialized before they are accessed. There are two lines of thought to initialization: Initialize all variables at the time the object is created (the traditional approach) or initialize at the time of first use. The first approach uses special member functions that are invoked when the object is first created, called constructors. Although this works, it often proves to be error prone. When adding a new variable you can easily forget to update the constructor(s). An alternative approach is called *lazy initialization* where fields are initialized by their getter member functions, as shown below. Notice the member function checks to see if the branch number is zero; if it is, then it sets it to the appropriate default value.

```
/**
    Answers the branch number, which is the leftmost
    four digits of the full account number.
    Account numbers are in the format BBBBAAAAAA.
 */
protected int getBranchNumber()

    if( branchNumber == 0)

        // The default branch number is 1000, which
        // is the main branch in downtown Bedrock.
        setBranchNumber( 1000);

    return branchNumber;
```

It is quite common to use lazy initialization for fields that are actually other objects stored in the database. For example, when you create a new inventory item you do not need to fetch whatever inventory item type from the database that you've set as a default. Instead, use lazy initialization to set this value the first time it is accessed so that you only have to read the inventory item type object from the database when and if you need it. This approach is advantageous for objects that have fields that aren't regularly accessed – why incur the overhead of retrieving something from persistent storage if you aren't going to use it?

Whenever lazy initialization is used in a getter member function you should document why the default value is what it is, as we saw in the example above. When you do this you take the mystery out of how fields are used in your code, improving both its maintainability and extensibility.

Field	Type	Getter name	Setter name
firstName	string	getFirstName()	setFirstName()
address	SurfaceAddress object	getAddress()	setAddress()
persistent	boolean	isPersistent()	setPersistent()
customerNumber	int	getCustomerNumber()	setCustomerNumber()
orderItems	Array of OrderItem objects	getOrderItems()	setOrderItems()

3.4bb Getters for Constants

The common Java wisdom is to implement constant values as static final fields. This approach makes sense for "constants" that are

guaranteed to be stable. For example, the class **Boolean** implements two *static final* fields called **TRUE** and **FALSE** which represents the two instances of that class. It would also make sense for a **DAYS_ IN_ A_ WEEK** constant whose value probably is never going to change.

However, many so- called business "constants" change over time because the business rule changes. Consider the following example: The Archon Bank of Cardassia (ABC) has always insisted that an account has a minimum balance of $500 if it is to earn interest. To implement this, we could add a static field named **MINIMUM_BALANCE** to the class Account that would be used in the methods that calculate interest. Although this would work, it isn't flexible. What happens if the business rules change and different kinds of accounts have different minimum balances, perhaps $500 for savings accounts but only $200 for checking accounts? What would happen if the business rule were to change to a $500 minimum balance in the first year, $400 in the second, $300 in the third, and so on? Perhaps the rule will be changed to $500 in the summer but only $250 in the winter? Perhaps a combination of all of these rules will need to be implemented in the future.

The point to be made is that implementing constants as fields isn't flexible, a much better solution is to implement constants as getter methods. In our example above, a static (class) method called **getMinimumBalance()** is far more flexible than a static field called **MINIMUM_ BALANCE** because we can implement the various business rules in this method and subclass it appropriately for various kinds of accounts.

Example 1.

```
/**
        Get the value of the account number.   Account numbers are in the following
        format: BBBBAAAAAA, where BBBB is the branch number and
        AAAAAA is the branch account number.
  */
  public long getAccountNumber()
  {
    return ( ( getBranchNumber() * 100000 ) + getBranchAccountNumber() );
   }

 /**
        Set the account number. Account numbers are in the following
        format: BBBBAAAAAA where BBBB is the branch number and
        AAAAAA is the branch account number.
  */
  public void setAccountNumber( int newNumber)
  {
    setBranchAccountNumber( newNumber % 1000000 );
    setBranchNumber( newNumber / 1000000 );
  }
```

Another advantage of constant getters is that they help to increase consistency of your code. Consider the code shown above – it doesn't work properly. An account number is the concatenation of the branch number and the branch account number. Testing our code, we find that the setter method, **setAccountNumber()** doesn't update branch account numbers properly (it takes the three leftmost digits, not four). That is because we used 1,000,000 instead of 100,000 to extract the field **branchAccountNumber**. Had we used a single source for this value, the constant getter, **getAccountNumberDivisor()** as we see in Example 2 below, our code would have been more consistent and would have worked.

Example 2.

```
/**
        Returns the divisor needed to separate the branch account number from the
        branch number within the full account number.
        Full account numbers are in the format BBBBAAAAAA.
   */
  public int getAccountNumberDivisor()
  {
    return ( (long) 1000000);
```

```
    }

/**
        Get the value of the account number.   Account numbers are in the following
        format: BBBBAAAAAA, where BBBB is the branch number and
        AAAAAA is the branch account number.
  */
  public long getAccountNumber()
  {
        return ( ( getBranchNumber() * getAccountNumberDivisor() ) +
                                        getBranchAccountNumber() );
    }

/**
        Set the account number. Account numbers are in the following
        format: BBBBAAAAAA where BBBB is the branch number and
        AAAAAA is the branch account number.
  */
  public void setAccountNumber( int newNumber)
  {
     setBranchAccountNumber( newNumber % getAccountNumberDivisor() );
     setBranchNumber( newNumber / getAccountNumberDivisor() );
    }
```

By using accessors for constants we decrease the chance of bugs and at the same time increase the maintainability of our system. When the layout of an account number changes, and we know that it eventually will (users are like that), chances are that our code will be easier to change because we've both hidden and centralized the information needed to build/ break up account numbers.

3.4bc  Accessors for Collections

The main purpose of accessors is to encapsulate the access to fields so as to reduce the coupling within your code. Collections, such as arrays and vectors, being more complex than single value fields naturally need to have more than just the standard getter and setter method implemented for them. In particular, because you can add and remove to and from collections, accessor methods need to be included to do so. Use this approach to add the following accessor methods where appropriate for a field that is a collection:

Method type	Naming convention	Example
Getter for the collection	getCollection()	getOrderItems()
Setter for the collection	setCollection()	setOrderItems()
Insert an object into the collection	insertObject()	insertOrderItem()
Delete an object from the collection	deleteObject()	deleteOrderItem()
Create and add a new object into the collection	newObject()	newOrderItem()
Getter for the object	getObject()	getOrderItem()
Setter for the object	setObject()	setOrderItem()

The advantage of this approach is that the collection is fully encapsulated, allowing you to later replace it with another structure, perhaps a linked list or a B- tree.

3.4bd  Accessing Several Fields Simultaneously

One of the strengths of accessor methods is that they enable you to enforce business rules effectively. Consider for example a class hierarchy of shapes. Each subclass of **Shape** knows its position via the use of two fields – **xPosition** and **yPosition** – and can be moved on the screen on a two- dimensional plane via invoking the method **move(Float xMovement, Float yMovement)**. In this

case, it doesn't make sense to move a shape along one axis at a time, instead it will move along both the x and the y axis simultaneously (it is acceptable to pass a value of 0.0 as for either parameter of the **move()** member function). The implication is that the **move()** method should be public, but the methods **setXPosition()** and **setYPosition()** should both be private, being invoked by the **move()** method appropriately.

An alternative implementation would be to introduce a setter method that updates both fields at once, as shown below. The methods **setXPosition()** and **setYPosition()** would still be private so that they may not be invoked directly by external classes or subclasses (you would want to add some documentation, shown below, indicating that they should not be directly invoked).

Remember that the naming convention for collections is to use a pluralized version of the type of information that it contains. Therefore a collection of order item objects would be called **orderItems**.

```
/**
        Set the position of the shape
*/
protected void setPosition( Float x, Float y)
{
        setXPosition( x);
        setYPosition( y);
}

/**
        Set the x position – Important: Invoke setPosition(), not this method.
*/
private void setXPosition( Float x)
{
        xPosition = x;
}

/**
        Set the y position of the shape
        Important: Invoke setPosition(), not this method.
*/
private void setYPosition( Float y)
{
        yPosition = y;
}
```

*Important note: This could have implemented with a single instance of **Point**, but it was done this way for an easy example.*

3.4c Visibility of Accessors

Always strive to make them protected, so that only subclasses can access the fields. Only when an 'outside class' needs to access a field should you make the corresponding getter or setter public. Note that it is common that the getter method be public and the setter protected.

Sometimes you need to make setters private to ensure certain invariants hold. For example, an **Order** class may have one field representing a collection of **OrderItem** instances and a second field called **orderTotal,** which is the total of the entire order. The **orderTotal** is a convenience field that is the sum or all sub- totals of the ordered items. The only methods that should update the value of **orderTotal** are those that manipulate the collection of order items. Assuming that those methods are all implemented in **Order**, you should make **setOrderTotal()** private, even though **getOrderTotal()** is more than likely public.

3.4d - Why Use Accessors?

Kanerva (1997) says it best: "Good program design seeks to isolate parts of a program from unnecessary, unintended, or otherwise unwanted outside influences. Access modifiers (accessors) provide an explicit and checkable means for the language to control such contacts." Accessor methods improve the maintainability of your classes in the following ways:

1. **Updating fields**. You have single points of update for each field, making it easier to modify and to test. In other words your fields are encapsulated.

2. **Obtaining the values of fields**. You have complete control over how fields are accessed and by whom.

3. **Obtaining the values of constants and the names of classes**. By encapsulating the value of constants and of class names in getter methods when those values/ names change you only need to update the value in the getter and not every line of code where the constant/ name is used.

4. **Initializing fields**. The use of lazy initialization ensures that fields are always initialized and that they are initialized only if they are needed.

5. **Reduction of the coupling between a subclass and its superclass(es)**. When subclasses access inherited fields only through their corresponding accessor methods, it makes it possible to change the implementation of fields in the superclass without affecting any of its subclasses, effectively reducing coupling between them. Accessors reduce the risk of the "fragile base class problem" where changes in a superclass ripple throughout its subclasses.

6. **Encapsulating changes to fields**. If the business rules pertaining to one or more fields change you can potentially modify your accessors to provide the same ability as before the change, making it easier for you to respond to the new business rules.

7. **Simplification of concurrency issues.** Lea (1997) points out that setter methods provide a single place to include a **notifyAll** if you have waits based on the value of that field. This makes moving to a concurrent solution much easier.

8. **Name hiding becomes less of an issue.** Although you should avoid name hiding, giving local variables the same names as fields, the use of accessors to always access fields means that you can give local variables any name you want – You do not have to worry about hiding field names because you never access them directly anyway.

3.4e - When Shouldn't You Use Accessors?

The only time that you might want to not use accessors is when execution time is of the utmost importance, however, it is a very rare case indeed that the increased coupling within your application justifies this action. Lea (1996) makes a case for minimizing the use of accessors on the grounds that it is often the case that the values of fields in combination must be consistent, and that it isn't wise to provide access to fields singly. He's right, so do not! However, Lea has missed the point that you do not need to make all accessor methods public. When you are in the situation that the values of some fields depend upon one another then you should introduce methods that do the "right thing" and make the appropriate accessor methods either protected or private as needed. You do not have to make all of your accessors public.

3.5 Always Initialize Static Fields

Doug Lea (1996) validly points out that you must ensure that static fields, also known as class fields, be given valid values because you can't assume that instances of a class will be created before a static field is accessed. Lea suggests the use of static initializers (Grand, 1997), static blocks of code which are automatically run when a class is loaded. Note that this is a problem only if you choose not to use accessor methods for static fields. With accessor methods you can always use lazy initialization to guarantee that the value of a field is set. The use of accessor methods to encapsulate fields gives you complete control over how they are used, while reducing coupling within your code; a win-win situation.

4. Standards for Local Variables

A local variable is an object or data item that is defined within the scope of a block, often a member function. The scope of a local variable is the block in which it is defined. The important coding standards for local variables focus on:

- Naming conventions
- Documentation conventions
- Declarations

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

- Streams
- Loop counters
- Exceptions

4.1a  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 (Gosling, Joy, Steele, 1996). For a stream used for both input and output, the implication is to use the name **inOut.**

4.1b  Naming Loop Counters

Because loop counters are a very common use for local variables, use names like **loopCounter** or simply **count**er. A possible problem with this approach is that you often find names like **counter1** and **counter2** in methods that require more than one counter.

4.1c  Naming Exception Objects

Because exception handling is also very common in Java coding the use of the letter **e** for a generic exception is considered acceptable (Gosling, Joy, Steele, 1996; Sandvik, 1996).

4.2 - Declaring and Documenting Local Variables

There are several conventions regarding the declaration and documentation of local variable in Java. These conventions are:

 **Declare one local variable per line of code.** This is consistent with one statement per line of code and makes it possible to document each variable with an inline comment (Vision, 2000).

 **Document local variables with an inline comment.** Inline commenting is a style in which a single line comment, denoted by //, immediately follows a command on the same line of code (this is called an endline comment). You should document what a local variable is used for and where appropriate why it is used, making your code easier to understand.

 **Declare local variables at the top of the class.** By declaring local variables at the top of the method, other programmers do not need to scroll through the code to find out what a local variable is used for.

Use local variables for one thing only. Whenever you use a local variable for more than one reason you effectively decrease its cohesion, making it difficult to understand. You also increase the chances of introducing bugs into your code from the unexpected side effects of previous values of a local variable. Yes, reusing local variables is more efficient because less memory needs to be allocated, but reusing local variables decreases the maintainability of your code and makes it more fragile. This usually isn't worth the small savings from not having to allocate more memory.

4.2a - General Comments about Declaration

Local variables that are declared between lines of code, for example within the scope of an if statement, can be difficult to find by people not familiar with your code.

One alternative to declaring local variables immediately before their first use is to instead declare them at the top of the code. Because your methods should be short anyway, see section 2.4e, it shouldn't be all that bad having to go to the top of your code to determine what the local variable is all about.

5. Standards for Parameters (Arguments) To Methods

The standards that are important for parameters/ arguments to methods focus on how they are named and how they are documented. Throughout this white paper the term *parameter* will refer to a method argument.

5.1  Naming Parameters

A standard taken from Smalltalk, is to use the naming conventions for local variables, with the addition of "a" or "an" on the front of the name. The addition of "a" or "an" helps to make the parameter stand out from local variables and fields, and avoids the name-hiding problem. This is the preferred approach.

Examples:

aCustomer
anInventoryItem
aPhotonTorpedo
anInputStream
anException

5.2  Documenting Parameters

Parameters to a method are documented in the header documentation for the method using the *javadoc @param* tag. You should describe:

1. **What it should be used for.** You need to document what a parameter is used for so that other developers understand the full context of how the parameter is used.

2. **Any restrictions or preconditions.** If the full range of values for a parameter is not acceptable to a method, then the invoker of that method needs to know. Perhaps a method only accepts positive numbers, or strings of less than five characters.

3. **Examples.** If it is not completely obvious what a parameter should be, then you should provide one or more examples in the documentation.

Tip – Use Interfaces for Parameter Types

Instead of specifying a class, such as **Object**, for the type of a parameter, specify an interface such as **Runnable**, if appropriate. The advantage is that this approach, depending on the situation, can be more specific (**Runnable** is more specific than **Object**), or may potentially be a better way to support polymorphism. Instead of insisting on a parameter being an instance of a class in a specific class hierarchy, you specify that it supports a specific interface implying that it only needs to be polymorphically compliant to what you need.

6. Standards for Classes, Interfaces, Packages and Source Code

This chapter concentrates on standards and guidelines for classes, interfaces, packages, and source codes. A class is a template from which objects are instantiated (created). Classes contain the declaration of fields and methods. Interfaces are the definition of a common signature, including both member functions and fields, which a class that implements an interface must support. A package is a collection of related classes. Finally, a source code is a source code file in which classes and interfaces are declared. Because Java allows source codes to be stored in a database an individual source code may not directly relate to a physical source code file.

6.1 - Standards for Classes

The standards that are important for classes are based on:

- Naming conventions
- Documentation conventions
- Declaration conventions
- The public and protected interface

6.1a  Naming Classes

The standard Java convention is to use a full English descriptor starting with the first letter capitalized using mixed case for the rest of the name (Gosling, Joy, Steele, 1996; Sandvik, 1996; Ambler, 1998a).

Examples:

Customer
Employee
Order
OrderItem
FileStream
String

6.1b  Documenting a Class

The following information should appear in the documentation comments immediately preceding the definition of a class:

1. **The purpose of the class.** Developers need to know the general purpose of a class so they can determine whether or not it meets their needs. It's also a good habit to document any important things to know about a class, for example is it part of a pattern or are there any interesting limitations to using it (Ambler, 1998a).

2. **Known bugs.** If there are any outstanding problems with a class they should be documented so that other developers understand the weaknesses/difficulties with the class. Furthermore, the reason for not fixing the bug should also be documented. Note that if a bug is specific to a single method then it should be directly associated with the method instead.

   Yes, it is better to fix bugs. However, sometimes you do not have the time to do so or it isn't important to your work at the moment. For example, you might know that a method would not work properly when passed a negative number, but that it does work properly for positive numbers. Your application only passes it positive numbers, so you can live with the bug but decide to be polite and document that the problem exists.

3. **The development/ maintenance history of the class.** It is common practice to include a history table listing dates, authors, and summaries of changes made to a class (Lea, 1996). The purpose of this is to provide maintenance programmers insight into the modifications made to a class in the past, as well as to document what has done what to a class.

4. **Document applicable invariants.** An invariant is a set of assertions about an instance or class that must be true at all "stable" times, where a stable time is defined as the period before a method is invoked on the object/ class and immediately after a method is invoked (Meyer, 1988). By documenting the invariants of a class you provide valuable insight to other developers as to how a class can be used.

5. **The concurrency strategy**. Any class that implements the interface **Runnable** should have its concurrency strategy fully described. Concurrent programming is a complex topic that is new for many programmers, therefore you need to invest the extra time to ensure that people can understand your work. It is important to document your concurrency strategy and why you chose that strategy over others. Common concurrency strategies (Lea, 1997) include the following: Synchronized objects, balking objects, guarded objects, versioned objects, concurrency policy controllers, and acceptors.

6.1c Class Declarations

One way to make your classes easier to understand is to declare them in a consistent manner. The common approach in Java is to declare a class in the following order (NPS, 1996):

public methods
public fields
protected methods
protected fields
private methods
private fields

Laffra (1997) points out that constructors and **finalize()** should be listed first, presumably because these are the first methods that another developer will look at first to understand how to use the class. Furthermore, because the standard is to declare all fields private, the declaration order really boils down to:

constructors
finalize()
public methods
protected methods
private methods
private fields

Within each grouping of methods it is common to list them in alphabetical order. Many developers choose to list the static methods within each grouping first, followed by instance methods, and then within each of these two sub- groupings list the methods alphabetically. Both of these approaches are valid, you just need to choose one and stick to it.

6.1d Minimize the Public and Protected Interface

One of the fundamentals of object- oriented design is to minimize the public interface of a class. There are several reasons for this:

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.

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.

3. **Greater flexibility.** This is directly related to coupling. Whenever you want to change the way that a method in your public interface is implemented, perhaps you want to modify what the method returns, 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.

It is clear that it is worth your while to minimize the public interface, but often what isn't so clear is that you also want to minimize the protected interface as well. The basic idea is that from the point of view of a subclass, the protected interfaces of all of its superclasses are effectively public – Any method in the protected interface can be invoked by a subclass. Therefore, you want to minimize the protected interface of a class for the same reasons that you want to minimize the public interface.

Tip – Define the Public Interface First

Most experienced developers define the public interface of a class before they begin coding it. First, if you do not know what services/ behaviors a class will perform, then you still have some design work to do. Second, it enables them to stub out the class quickly so that other developers who rely on it can at least work with the stub until the "real" class has been developed. Third, this approach provides you with an initial framework around which to build your class.

6.2 - Standards for Interfaces

The standards that are important for interfaces are based on:

- Naming conventions
- Documentation conventions

6.2a  Naming Interfaces

The Java convention is to name interfaces using mixed case with the first letter of each word capitalized. 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**, are also common (Gosling, Joy, Steele, 1996). It is customary to postfix the name with either 'able,' 'ible' or 'er' but this is not required.

Prefix the letter 'I' to the interface name. Coad and Mayfield (1997) suggest appending the letter 'I' to the front of an interface names, resulting in names like **ISingleton** or **IRunnable**. This approach helps to distinguish interface names from class and package names. The advantage of this naming convention is that it makes your class diagrams, sometimes referred to as object models, easier to read. The main disadvantage is that the existing interfaces, such as **Runnable**, aren't named using this approach. This interface naming convention is also popular for Microsoft's COM/ DCOM architecture.

6.2b  Documenting Interfaces

The following information should appear in the documentation comments immediately preceding the definition of an interface:

1. **The purpose.** Before other developers will use an interface, they need to understand the concept that it encapsulates. In other words, they need to know its purpose. A really good test of whether or not you need to define an interface is whether or not you can easily describe its purpose. If you have difficulties describing it, then chances are pretty good you do not need the interface to begin with. Because the concept of interfaces is new to Java, people are not yet experienced in their appropriate use and they are likely to overuse them because they are new. Just like the concept of inheritance, and in particular multiple inheritance, was greatly abused by developers new to object- orientation, it's likely that interfaces will also be greatly abused at first by programmers new to Java.

2. **How it should and shouldn't be used.** Developers need to know both how an interface is to be used, as well as how it shouldn't be used (Coad & Mayfield, 1997).

Because the signature for methods is defined in an interface, for each method signature you should follow the method documentation conventions discussed in chapter 2.

6.3 - Standards for Packages

The standards that are important for packages are based on:

- Naming conventions
- Documentation conventions

6.3a  Naming Packages

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

1. **Identifiers are separated by periods**. To make package names more readable, Sun suggests that the identifiers in package names be separated by periods. For example, the package name **java.awt** is comprised of two identifiers, **java** and **awt** .

2. **The standard java distribution packages from Sun begin with the identifier 'java'.** 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.

3. **Package names begin with the reversed Internet domain name for your organization.** A modified approach has been adopted due to the likelihood that WisDOT's Internet domain name may soon be changed. Each package name will follow the pattern in this example: **dot. project. mgt. it . busnobj. YourClassName.class**

6.3b  Documenting a Package

You should maintain one or more external documents that describe the purpose of the packages developed by your organization. For each package you should document:

1. **The rationale for the package.** Other developers need to know what a package is all about so that they can determine whether or not they want to use it, and if it is a shared package whether or not they want to enhance or extend it.

2. **The classes in the package.** Include a list of the classes and interfaces in the package with a brief, one- line description of each so that other developers know what the package contains.

Lea (1996) suggests creating an HTML file called *index.html* for each package, putting the file into the appropriate directory for the package. A better name would be the fully qualified name of the package, postfixed with *.html* , so that you do not have to worry about accidentally overwriting one package documentation file with another.

6.4 - Standards for Source Code Files

6.4a  Naming Source Code Files

Source code files, should be given the name of the primary class or interface that is declared within it. Use the same name for the package/ class for the file name, using the same case. The extension **.java** should be postfixed to the file name. Visual Age for Java enforces this standard automatically.

Examples:

Customer. java
Singleton. java
SavingsAccount. java

7. Miscellaneous Standards/Issues

This chapter covers several standards/guidelines that are important but don't fit into any other category.

7.1  Reuse

Any Java class library or package that you purchase/reuse from an external source should be certified as 100% pure Java (Sun, 1997). By enforcing this standard you are guaranteed that what you are reusing will work on all platforms that you choose to deploy it on. You can obtain Java classes, packages, or applets from either a third- party development company that specializes in Java

libraries or another division or project team within your organization.

Tip – 100% Pure is 100% Dead On

In most people's opinion, the 100% Pure effort from Sun is exactly what Java needs. There are two portability issues with Java: source code portability and bytecode portability. Developers who ported from JDK 1.0 to JDK 1.1 now have a better appreciation for those two issues. In many ways, the 100% Pure effort is Sun's recognition that portability doesn't come free just because you use Java; you actually have to work at it to ensure that your code is portable. With multiple vendors of Java, several of who would like to mold Java in their own image, without something like the 100% Pure effort Java code will become just as portable as C code – not very.

Sun's message to the other Java vendors and to Java developers is clear: Proprietary Java solutions will not be tolerated (Ambler, 1998a).

7.2 - Optimizing Java Code

Optimizing code is one of the last things that programmers should be thinking about, not one of the first. You'll want to leave optimization to the end because you want to optimize only the code that needs it – very often a small percentage of your code results in the vast majority of the processing time, and this is the code that you should be optimizing. A mistake made by inexperienced programmers is to try to optimize all of their code, even code that already runs fast enough. It's preferable to optimize the code that needs it and then move on to more things other than trying to squeeze out every single CPU cycle. **Do not waste your time optimizing code that no one cares about.**

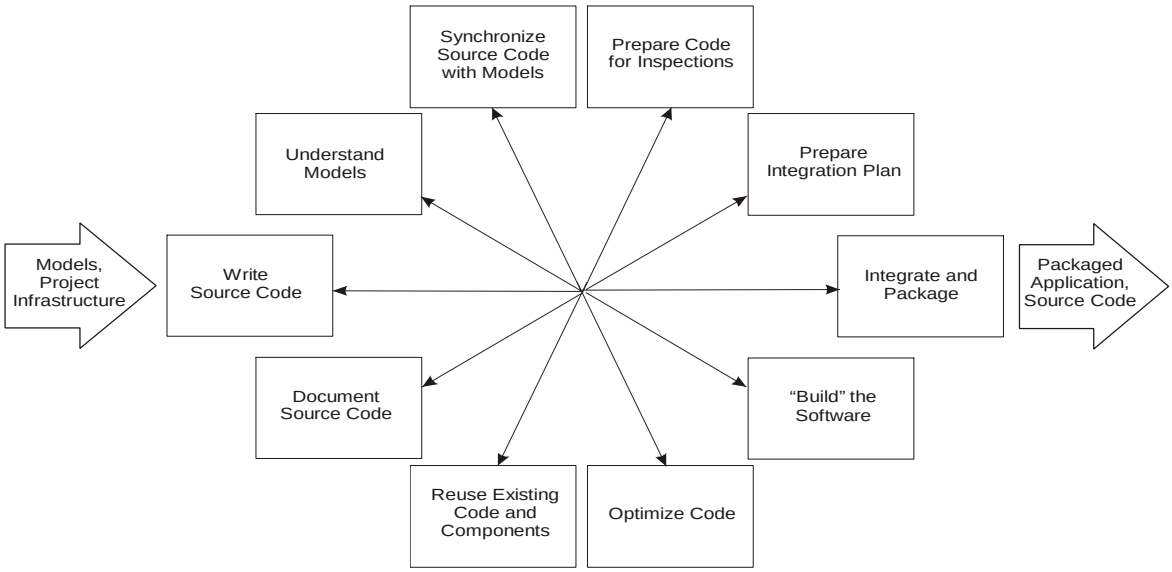

Figure 1. The Program process pattern.
Figure 1 presents the Program stage process pattern (Ambler, 1998b) which describes the iterative process by which you develop source code. This process pattern shows that code optimization is a part of programming, but only one of many parts. This is advice that all coders should take to heart.

Tip – Define Your Project's Development Priorities

Everybody has their own idea about what is important, and software developers are no different. The issue is that your project, and your organization, needs to define what their development priorities are so that all team members are working to the same vision. Maguire (1994) believes that your organization needs to establish a ranking order for the following factors: size, speed, robustness, safety, testability, maintainability, simplicity, reusability, and portability. These factors define the quality of the software that

your produce, and by prioritizing them you will help to define the development goals for your team and will reduce the opportunities for disagreement between your developers.

What should you look for when optimizing code? Koenig (1997) points out that the most important factors are fixed overhead and performance on large inputs. The reason for this is simple: fixed overhead dominates the runtime speed for small inputs and the algorithm dominates for large inputs. His rule of thumb is that a program that works well for both small and large inputs will likely work well for medium-sized inputs.

Developers who have to write software that work on several hardware platforms and/ or operating systems need to be aware of idiosyncrasies in the various platforms. Operations that might appear to take a particular amount of time, such as the way that memory and buffers are handled, often show substantial variations between platforms. It is common to find that you need to optimize your code differently depending on the platform.

Another issue to be aware of when optimizing code is the priorities of your users depending on the context people will be sensitive to particular delays. For example, your users will likely be happier with a screen that draws itself immediately and then takes eight seconds to load data than with a screen that draws itself after taking five seconds to load data. In other words most users are willing to wait a little longer as long as they're given immediate feedback, important knowledge to have when optimizing your code. **You do always need to make your code run faster to optimize it in the eyes of your users.**

Although optimization may mean the difference between the success and failure of your application, never forget that it is far more important to get your code to work properly. Never forget that s low software that works is always preferable to fast software that does not.

7.3  Writing Java Test Harnesses

Object- oriented testing is critical topic that has been all but ignored by the object development community. The reality is that either you or someone else will have to test the software that you write, regardless of the language that you've chosen to work in. A test harness is the collection of methods, some embedded in the classes themselves (this is called built- in tests) and some in specialized testing classes, that are used to test your application.

1. **Prefix all testing method names with 'test'.** This allows you to quickly find all the testing methods in your code. The advantage of prefixing the name of test methods with 'test' is that it allows you to easily strip your testing methods out of your source code before compiling the production version of it.

2. **Name all method test methods consistently.** Method testing is the act of verifying that a single method performs as defined. All method test member functions should be named following the format '**testMemberFunctionNameForTestName**'. For example, the test harness methods to test **withdrawFunds()** would include **testWithdrawFundsForInsufficientFunds()** and **testWithdrawFundsForSmallWithdrawal()** . If you have a series of tests for **withdrawFunds()** you may choose to write a method called **testWithdrawFunds()** that invokes all of them.

3. **Name all class test methods consistently.** Class testing is the act of verifying that a single class performs as defined. All class test methods should be named following the format ' **testSelfForTestName** '. For example, the test harness methods to test the Account class **testSelfForSimultaneousAccess()** and **testSelfForReporting()**.

4. **Create a single point for invoking the tests for a class.** Develop a static method called **testSelf()** that invokes all class testing and method testing methods.

5. **Document your test harness methods.** The documentation should include a description of the test as well as the expected results of the test. If you choose to document your tests in an external document, such as a master test/QA plan (Ambler, 1998b; Ambler, 1999) then refer to the appropriate section of that plan in your source code documentation to support traceability.

Further Reading In Object-Oriented Testing

If you’re interesting in learning more about object- oriented testing then you might want to check out the following resources:

- Full Lifecycle Object- Oriented Testing (Ambler, 1998a; Ambler, 1998b), also known as FLOOT
- Test In the Small process pattern (Ambler, 1998b)
- Test In The Large process pattern (Ambler, 1999)
- *http:// www.ambysoft.com/ booksTesting.html*
- The Process Patterns Page (*http://www ambysoft.com/processPatternsPage htm*)

8. The Secrets of Success

This white paper exists to help make Java developers more productive. The bad news is that having a standards document in your possession doesn't automatically make you more productive as a developer. To be successful you must choose to become more productive, and that means you must apply these standards effectively.

8.1 - Using These Standards Effectively

The following words of advice will help you to use the Java coding standards and guidelines described in this white paper more effectively:

1. **Understand the standards.** Take the time to understand why each standard and guideline leads to greater productivity. For example, do not declare each local variable on its own line just because it's written in this white paper; do it because you understand that it increases the understandability of your code.

2. **Believe in them.** Understanding each standard is a start, but you also need to believe in them too. Following standards shouldn't be something that you do when you have the time, it should be something that you always do because you believe that this is the best way to code. Using these tools and techniques that makes most developers more productive. Intelligent standards applied appropriately lead to significant increases in developer productivity.

3. **Follow them while you are coding, not as an afterthought.** Documented code is easier to understand while you are writing it as well as after it is written. Consistently named methods and fields are easier to work with during development as well as during maintenance. Clean code is easier to work with during development and during maintenance. The bottom line is that following standards will increase your productivity while you are developing as well as make your code easier to maintain (hence making maintenance developers more productive too). Too many people write sloppy code while they are developing, and then spend almost as long cleaning it up at the end so that it will pass inspection. That is wasteful. If you write clean code right from the beginning you can benefit from it while you are creating it. That is smart.

4. **Make them part of your quality assurance process.** Part of a code inspection should be to ensure that source code follows the standards adopted by your organization. Use standards as the basis from which you train and mentor your developers to become more effective.

5. **Adopt the standards that make the most sense for you.** You do not need to adopt every standard at once, instead start with the ones that you find the most acceptable, or perhaps the least unacceptable, and then go from there. Bring standards into your organization in stages, slowly but surely.

8.2 - Other Factors That Lead to Successful Code

I'd like to share several techniques with you from *Building Object Applications That Work* (Ambler, 1998a) that, in addition to following standards, lead to greater productivity:

1. **Program for people, not the machine.** The primary goal of your development efforts should be that your code is easy for other people to understand. If no one else can figure it out, then it isn't any good. Use naming conventions. Document your code. Paragraph it.

2. **Design first, then code.** Have you ever been in a situation where some of the code that your program relies on needs to be changed? Perhaps a new parameter needs to be passed to a method, or perhaps a class needs to be broken up into several classes. How much extra work did you have to do to make sure that your code works with the reconfigured version of the modified code? How happy were you? Did you ask yourself why somebody didn't stop and think about it first when he or she originally wrote the code so that this didn't need to happen? That they should have DESIGNED it first? Of course you did. If you take the time to figure out how you are going to write your code before you actually start coding you'll probably spend less time writing it. Furthermore, you'll potentially reduce the impact of future changes on your code simply by thinking about them up front.

3. **Develop in small steps.** I have always found that developing in small steps, writing a few member functions, testing them, and then writing a few more methods is often far more effective than writing a whole bunch of code all at once and then trying to fix it. It is much easier to test and fix ten lines of code than 100. In fact, I would safely say that you could program, test, and fix 100

lines of code in ten 10- line increments in less than half the time than you could write a single one- hundred line block of code that did the same work. The reason for this is simple. Whenever you are testing your code and you find a bug you almost always find the bug in the new code that you just wrote, assuming of course that the rest of the code was pretty solid to begin with. You can hunt down a bug a lot faster in a small section of code than in a big one. By developing in small incremental steps you reduce the average time that it takes to find a bug, which in turn reduces your overall development time.

4. **Read, read, read.** This industry moves far too quickly for anyone to coast on previous successes. In fact, people within Sun estimate that it's a full time job for two to three people just to keep up with what's happening with Java, let alone what's happening in the object-orientation field or even development in general. That implies that you need to invest at least some time trying to keep up. There is a suggested reading list indicating what are considered the key development books at the end of this white paper.

5. **Work closely with your users.** Good developers work closely with their users. Users know the business. Users are the reason why developers create systems, to support the work of users. Users pay the bills, including the salaries of developers. You simply can't develop a successful system if you do not understand the needs of your users, and the only way that you can understand their needs is if you work closely with them.

6. **Keep your code simple.** Complex code might be intellectually satisfying to write but if other people can't understand it then it isn't any good. The first time that someone, perhaps even you, is asked to modify a piece of complex code to either fix a bug or to enhance it chances are pretty good that the code will get rewritten. In fact, you've probably even had to rewrite somebody else's code because it was too hard to understand. What did you think of the original developer when you rewrote their code; did you think that person was a genius or an annoyance? Writing code that needs to be rewritten later is nothing to be proud of, so follow the KISS rule: Keep it simple, stupid.

7. **Learn common patterns, antipatterns, and idioms.** There is a wealth of analysis, design, and process patterns and antipatterns, as well as programming idioms, available to guide you in increasing your development productivity. Patterns provide the opportunity for very high levels of reuse within your software development projects (Ambler, 1998b). For more information, visit The Process Patterns Resource Page (*http://www ambysoft com/processPatternsPage.html*) for links to key patterns resources and process-oriented web sites.

Sun has introduced something called "doclets" which give you the ability to extend javadoc (i.e. add new tags). For more information about doclets, and about any proposed new tags, please refer to the information posted at **http://www.javasoft.com:81/products/jdk/1.2/docs/tooldocs/javadoc**.

9. Summary

In this white paper many standards and guidelines for Java developers have been detailed. Because this white paper is reasonably large they have have summarized them here for your convenience.

This chapter is organized into several one- page summaries of our Java coding standards, collected by topic. These topics are:

- Java naming conventions
- Java documentation conventions
- Java coding conventions

Before we summarize the rest of the standards and guidelines described in this white paper, I would like to reiterate the prime directive:

When you go against a standard, document it! All standards, except for this one, can be broken. If you do so, you must document why you broke the standard, the potential implications of breaking the standard, and any conditions that may/must occur before the standard can be applied to this situation.

9.1 - Java Naming Conventions

With a few exceptions discussed below, you should always use full English descriptors when naming things. Furthermore, you should use lower case letters in general, but capitalize the first letter of class names and interface names, as well as the first letter of any non-initial word.

General Concepts:

- Use full English descriptors
- Use terminology applicable to the domain
- Use mixed case to make names readable
- 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

Item	Naming Convention	Example
Arguments/ parameters	Use a full English description of value/ object being passed, prefixing the name with ‘a’ or ‘an. ’ .	aCustomer, anAccount
Fields/ fields/ properties	Use a full English description of the field, with the first letter in lower case and the first letter of any non- initial word in uppercase.	firstName, lastName, warpSpeed
Boolean getter methods	All boolean getters must be prefixed with the word ‘is.’ If you follow the naming standard for boolean fields described above then you simply give it the name of the field.	isPersistent(),isString(), isCharacter()
Classes	Use a full English description, with the first letters of all words capitalized.	Customer, SavingsAccount
Source code files	Use the name of the class or interface, or if there is more than one class in the file than the primary class, prefixed with ‘.java’ to indicate it is a source code file.	Customer. java, SavingsAccount. java, Singleton. java
Components/ widgets	Use a full English description that describes what the component is used for with the type of the component concatenated onto the end.	OkButton, customerList, fileMenu,
Constructors	Use the name of the class.	Customer(), SavingsAccount()
Destructors	Java does not have destructors, but instead will invoke the finalize() method before an object is garbage collected.	finalize()
Exceptions	It is generally accepted to use the letter ‘e’ to represent exceptions.	e

Final static fields (constants)	Use all uppercase letters with the words separated by underscores. A better approach is to use final static getter methods because it greatly increases flexibility.	MIN_ BALANCE, DEFAULT_ DATE
Getter methods	Prefix the name of the field being accessed with 'get.'	getFirstName(),getLastName( ),getWarpSpeeed()
Interfaces	Use a full English description describing the concept that the interface encapsulates, with the first letters of all words capitalized. Prefix the letter 'I' to the interface name. It is customary to postfix the name with either 'able,' 'ible,' or 'er' but this is not required.	IRunnable, IContactable, IPrompter, ISingleton
Local variables	Use full English descriptions with the first letter in lower case but do not hide existing fields/ fields. For example, if you have a field named 'firstName' do not have a local variable called 'firstName.'	grandTotal, customer, newAccount
Loop counters	Use names such as loopCounter.	loopCounter1, loopCounter2, etc
Packages	Use full English descriptions, using lower case. For global packages, reverse the name of your Internet domain and concatenate to this the package name. Each package name will follow this pattern, (words in ***italics*** would be replaced with application specific names); **dot.*busnfunction.busnfunctionqualifier.subject.objectgroup.classname***	**dot.project.mgt.it. busnobj. YourClassName.class**
Methods	Use a full English description of what the method does, starting with an active verb whenever possible, with the first letter in lower case.	openFile addAccount()(),
Setter methods	Prefix the name of the field being accessed with 'set'.	setFirstName(),setLastName( ),setWarpSpeed()

9.2 - Java Documentation Conventions

A really good rule of thumb to follow regarding documentation is to ask yourself if you've never seen the code before, what information would you need to effectively understand the code in a reasonable amount of time.

General Concepts:

- Comments should add to the clarity of your code
- If your program isn't worth documenting, it probably isn't 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

9.2a - Java Comment Types

The following chart describes the three types of Java comments and standard uses for them.

Comment Type	Usage	Example
Documentation	Use documentation comments immediately before declarations of interfaces, classes, methods, and fields to document them. Documentation comments are processed by javadoc (see below) to create external documentation for a class.	/** Customer – A customer is any person or organization that we sell services and products to. @author S. W. Ambler */

C style	Use C- style comments to document out lines of code that are no longer applicable, but that you want to keep just in case you users change their minds, or because you want to temporarily turn it off while debugging.	/* This code was commented out by J. T. Kirk on Dec 9, 1997 because it was replaced by the preceding code. Delete it after two years if it is still not applicable. . . . (the source code) */

Single line	Use single line comments internally within methods to document business logic, sections of code, and declarations of temporary variables.	// Apply a 5% discount to all invoices
		// over $1000 as defined by the Sarek
		// generosity campaign started in
		// Feb. of 1995.

9.2b - What To Document

The following chart summarizes what to document regarding each portion of Java code that you write.

Item	What to Document
Arguments/ parameters	The type of the parameter What it should be used for Any restrictions or preconditions Examples
Fields/ fields/ properties	Its description Document all applicable invariants Examples Concurrency issues Visibility decisions
Classes	The purpose of the class Known bugs The development/ maintenance history of the class Document applicable invariants The concurrency strategy
Source codes	Each class/ interface defined in the class, including a brief description The file name and/ or identifying information Copyright information
Getter method	Document initialization.
Interfaces	The purpose How it should and shouldn't be used
Local variables	Its use/ purpose
Methods – Documentation	What and why the method does what it does What a method must be passed as parameters What a method returns Known bugs Any exceptions that a method throws Visibility decisions How a method changes the object Include a history of any code changes Examples of how to invoke the method if appropriate Applicable preconditions and postconditions Document all concurrency

Methods – Internal comments	Control structures Why, as well as what, the code does Local variables Difficult or complex code The processing order
Package	The rationale for the package The classes in the package

9.3 - Java Coding Conventions (General)

There are many conventions and standards that are critical to the maintainability and enhancability of your Java code. 99.9% of the time it is more important to program for people, your fellow developers, than it is to program for the machine. Making your code understandable to others is of utmost importance.

Convention Target	Convention
Accessor methods	Consider using lazy initialization for fields in the database Initialize variables before they are accessed Use accessors for obtaining and modifying all fields Use accessors for 'constants" For collections, add methods to insert and remove items Whenever possible, make accessors protected, not public
Fields	Fields should always be declared private Do not directly access fields, instead use accessor methods Do not use final static fields (constants), instead use accessor methods Do not hide names Always initialize static fields
Classes	Minimize the public and protected interfaces Define the public interface for a class before you begin coding it Declare the fields and methods of a class in the following order: • constructors • finalize() • public methods • protected methods • private methods • private field
Local variables	Do not hide names Declare one local variable per line of code Document local variables with an inline comment Declare local variables immediately before their use Use local variables for one thing only
Methods	Document your code Paragraph your code Use white space, one line before control structures and two before method declarations A method should be understandable in less than thirty seconds Write short, single command lines Restrict the visibility of a method as much as possible Specify the order of operations

Glossary

100% pure – Effectively a "seal of approval" from Sun that says that a Java applet, application or package, will run on ANY platform which supports the Java VM.

Accessor – A method that either modifies or returns the value of a field. Also known as an access modifier. See getter and setter.

Analysis pattern – A modeling pattern that describes a solution to a business/domain problem.

Antipattern – An approach to solving a common problem, an approach that in time proves to be wrong or highly ineffective.

Argument – See parameter.

BDK – Beans development kit.

Block – A collection of zero or more statements enclosed in (curly) braces.

Braces – The characters **{** and **}**, known as an open brace and a close brace respectively, are used to define the beginning and end of a block. Braces are informally referred to as 'curlies'.

Class – A definition, or template, from which objects are instantiated.

Class testing – The act of ensuring that a class and its instances (objects) perform as defined.

Component – An interface widget such as a list, button, or window.

Constant getter – A getter method which returns the value of a "constant," which may in turn be hard coded or calculated if need be.

Constructor – A method which performs any necessary initialization when an object is created.

Containment – An object contains other objects that it collaborates with to perform its behaviors. This can be accomplished either the use of inner classes (JDK 1.1+) or the aggregation of instances of other classes within an object (JDK 1.0+).

CPU – Central processing unit.

C- style comments – A Java comment format, /* … */, adopted from the C/ C++ language that can be used to create multiple- line comments. Commonly used to "document out" unneeded or unwanted lines of code during testing.

Design pattern – A modeling pattern that describes a solution to a design problem.

Destructor – A C++ class method that is used to remove an object from memory once it is no longer needed. Because Java manages its own memory, this kind of method is not needed. Java does, however, support a method called **finalize()** that is similar in concept.

Documentation comments – A Java comment format, /** … */, that can be processed by *javadoc* to provide external documentation for a class file. The main documentation for interfaces, classes, methods, and fields should be written with documentation comments.

Field – A variable, either a literal data type or another object, that describes a class or an instance of a class. Instance fields describe objects (instances) and static fields describe classes. Fields are also referred to as field variables, properties and attributes of a class.

finalize() – A method that is automatically invoked during garbage collection before an object is removed from memory. The purpose of this method is to do any necessary cleanup, such as the closing of open files.

Garbage collection – The automatic management of memory where objects that are no longer referenced are automatically removed from memory.

Getter – A type of accessor method that returns the value of a field. A getter can be used to answer the value of a constant, which is often preferable to implementing the constant as a static field because this is a more flexible approach.

HTML – Hypertext markup language, an industry- standard format for creating web pages.

Indenting – See paragraphing.

Inline comments – The use of a line comment to document a line of source code where the comment immediate follows the code on the same line as the code. Single line comments are typically used for this, although C- style comments can also be employed.

Interface – The definition of a common signature, including both methods and fields, which a class that implements an interface must support. Interfaces promote polymorphism by composition.

I/ O – Input/ output.

Invariant – A set of assertions about an instance or class that must be true at all "stable" times, the periods before and after the invocation of a method on the object/ class.

Java – An industry- standard object- oriented development language that is well- suited for developing applications for the Internet and applications that must operate on a wide variety of computing platforms.

javadoc – A utility included in the JDK that processes a Java source code file and produces an external document, in HTML format, describing the contents of the source code file based on the documentation comments in the code file.

JDK – Java development kit.

Lazy initialization – A technique in which a field is initialized in its corresponding getter member function the first time that it is needed. Lazy initialization is used when a field is not commonly needed and it either requires a large amount of memory to store or it needs to be read in from permanent storage.

Local variable – A variable that is defined within the scope of a block, often a method. The scope of a local variable is the block in which it is defined.

Master test/quality assurance (QA) plan – A document that describes your testing and quality assurance policies and procedures, as well as the detailed test plans for each portion of your application.

Method – A piece of executable code that is associated with a class, or the instances of a class. Think of a method as the object-oriented equivalent of a function.

Method signature – See signature.

Method testing – The act of ensuring that a method performs as defined.

Modeling pattern – A pattern depicting a solution, typically in the form of a class model, to a common modeling problem.

Name hiding – This refers to the practice of using the same, or at least similar, name for a field/ variable/ argument as for one of higher scope. The most common abuses of name hiding are to name a local variable the same as an instance field or to make the parameter name the same as the object's variable's name in a method. Name hiding should be avoided as it makes your code harder to understand and prone to bugs.

Overload – A method is said to be overloaded when it is defined more than once in the same class (or in a subclass), the only difference being the signature of each definition.

Override – A method is said to be overridden when it is redefined in a subclass and it has the same signature as the original definition.

Package – A collection of related classes.

Paragraphing – A technique where you indent the code within the scope of a code block by one unit, usually a horizontal tab, so as to distinguish it from the code outside of the code block. Paragraphing helps to increase the readability of your code.

Parameter – An argument passed to a method. A parameter may be a defined type, such as a string or an int, or an object.

Pattern – The description of a general solution to a common problem or issue from which a detailed solution to a specific problem may be determined. Software development patterns come in many flavors, including but not limited to analysis patterns, design patterns, and process patterns.

Postcondition – A property or assertion that will be true after a method is finished running.

Precondition – A constraint under which a method will function properly.

Process pattern – A pattern that describes a proven, successful approach and/ or series of actions for developing software.

Property – See field.

Quality assurance (QA) – The process of ensuring that the efforts of a project meet or exceed the standards expected of them.

Setter – An accessor method that sets the value of a field.

Signature – The combination of the method's name, the type of parameters, if any, and their order that must be passed to a method. Also called the method signature.

Single-line comments – A Java comment format, // … , adopted from the C/ C++ language that is commonly used for the internal method documentation of business logic.

Source code – A source code file, either a physical one on disk or a "virtual" one stored in a database, in which classes and interfaces are declared.

Tags – A convention for marking specified sections of documentation comments that will be processed by *javadoc* to produce professional-looking comments. Examples of tags include *@see* and *@author* .

Test harness – A collection of methods for testing your code

UML – Unified modeling language, an industry standard modeling notation.

Visibility – A technique that is used to indicate the level of encapsulation of a class, method, or field. The

keywords public, protected, and private can be used to define visibility.

White space – Blank lines, spaces, and tabs added to your code to increase its readability.

Widget – See component.

References and Suggested Reading

Ambler, S. (1995). *The Object Primer: The Application Developer's Guide to Object Orientation* . New York: SIGS Books/ Cambridge University Press.

Ambler, S. W. (1998a) . *Building Object Applications That Work: Your Step- By- Step Handbook for Developing Robust Systems with Object Technology* . New York: SIGS Books/ Cambridge University Press.

Ambler, S. W. (1998b) . *Process Patterns: Building Large- Scale Systems Using Object Technology* . New York: SIGS Books/ Cambridge University Press.

Ambler, S. W. (1999) . *More Process Patterns: Delivering Large- Scale Systems Using Object Technology* . New York: SIGS Books/ Cambridge University Press.

Campione, M and Walrath, K (1998). *The Java Tutorial Second Edition: Object- Oriented Programming for the Internet* . Reading, MA: Addison Wesley Longman Inc.

Chan, P. and Lee, R. (1997). *The Java Class Libraries: An Annotated Reference* . Reading, MA: Addison Wesley Longman Inc.

Coad, P. and Mayfield, M. (1997). *Java Design: Building Better Apps & Applets* . Upper Saddle River, NJ: Prentice Hall Inc.

DeSoto, A. (1997). *Using the Beans Development Kit 1.0 February 1997: A Tutorial* . Sun Microsystems. Gosling, J., Joy, B., Steele, G. (1996). *The Java Language Specification* . Reading, MA: Addison Wesley Longman Inc.

Grand, M. (1997). *Java Language Reference* . Sebastopol, CA: O'Reilly & Associates, Inc. Heller, P. and Roberts, S. (1997). *Java 1.1 Developer's Handbook* . San Francisco: Sybex Inc.

Kanerva, J. (1997). *The Java FAQ* . Reading, MA: Addison Wesley Longman Inc.

Koenig, A. (1997). *The Importance – and Hazards – of Performance Measurement* . New York: SIGS Publications, Journal of Object- Oriented Programming, January, 1997, 9(8), pp. 58- 60.

Laffra, C. (1997). *Advanced Java: Idioms, Pitfalls, Styles and Programming Tips* . Upper Saddle River, NJ: Prentice Hall Inc.

Lea, D. (1996). *Draft Java Coding Standard* . http:// g. oswego. edu/ dl/ html/ javaCodingStd. html

Lea, D. (1997). *Concurrent Programming in Java: Design Principles and Patterns* . Reading, MA: Addison Wesley Longman Inc.

McConnell, S. (1993). *Code Complete – A Practical Handbook of Software Construction* . Redmond, WA: Microsoft Press.

McConnell, S. (1996). *Rapid Development: Taming Wild Software Schedules* . Redmond, WA: Microsoft Press.

Meyer, B. (1988). *Object- Oriented Software Construction* . Upper Saddle River, NJ: Prentice Hall Inc.

Meyer, B. (1997). *Object- Oriented Software Construction, Second Edition* . Upper Saddle River, NJ: Prentice- Hall PTR.

Nagler, J. (1995). *Coding Style and Good Computing Practices* . http:// wizard. ucr. edu/~ nagler/ coding_ style. html

NPS (1996). *Java Style Guide* . United States Naval Postgraduate School. http:// dubhe. cc. nps. navy. mil/~ java/ course/ styleguide. html

Niemeyer, P. and Peck, J. (1996). *Exploring Java* . Sebastopol, CA: O'Reilly & Associates, Inc.

Sandvik, K. (1996). *Java Coding Style Guidelines* . http:// reality. sgi. com/ sandvik/ JavaGuidelines. html

Sun Microsystems (1996). *javadoc – The Java API Documentation Generator* . Sun Microsystems.

Sun Microsystems (1997). *100% Pure Java Cookbook for Java Developers – Rules and Hints for Maximizing the Portability of Java Programs*. Sun Microsystems.

Vanhelsuwe, L. (1997). *Mastering Java Beans* . San Francisco: Sybex Inc.

Vision 2000 CCS Package and Application Team (1996). *Coding Standards for C, C++, and Java* . http:// v2ma09. gsfc. nasa. gov/ coding_ standards. html

Index

@

@author tag……. 4
@deprecated tag……. 4
@exception tag…… 4
  usage…… 7
@param tag……. 4
  usage…… 22
@return tag……. 4
@since tag…… 4
@version tag…… 4

1

100% pure…… 27
  definition……. 36

A

Abbreviation, usage…….. 1
Accessor method…… 13
  advantages and disadvantages…….. 18
  constant getters……. 15
  definition……. 36
  for collections 16
  getters.....…….. 5
  setter methods……... 6
  visibility…….. 18
Analysis pattern, definition…… 36
Antipattern, definition……. 36
Argument……...See parameter
Attribute
  accessors……. 13
  definition……. 36
  documentation……... 12
  getter…… 13
  getter methods……... 13
  initialization……... 14
  name hiding…… 12
  naming conventions…….. 11
  setter…… 13
  setter methods 13
  visibility…….. 12

B

Banner comments……..… 2
BDK, definition…….. 36
Block, definition……. 36
Braces
  definition……. 36
  documentation of……... 9

C

Class……..…… 23
  definition……. 36
  documentation conventions…… 23
  naming conventions…….. 23
Class testing, definition……... 36
Clean code……... 9
Code clarity……. 3
Code history…… 7
Code inspections…… 30
Coding standards
  classes…….. 23
  for local variables…….. 20
  for methods 5
  importance of……. 1
  interfaces……. 25
  packages…….. 26
  parameters……. 22
  using them effectively…….. 30
Collections
  accessor methods……... 16
Comments
  banner style…… 2
  in methods…….. 8
  types of……. 2
Complex code, documenting…… 8
Component, definition…….. 36
  naming conventions……. 11
Concurrency
  documentation of…….8, 13, 24
Constant getter…….. 15
  definition……. 36
Constants, naming conventions……. 11
Constructor, definition……. 36
Containment, definition…….. 36
Control Structures…… 8
Coupling
  and accessors……. 16
  and inheritance……. 17
C-style
  comments…… 3
  definition……. 36
  usage…… 8

D

Design first……..… 28
Design pattern, definition…… 36
Destructor, definition…… 36
Develop in small steps…….. 31
doclets…….. 31

Documentation
clarity…….. 2
code history…… 7
concurrency…….…….8, 13, 24
internal…… 8
of attributes…… 12
of closing braces……. 9
of complex code…….. 8
of local variables…… 20
of methods…….. 7
of processing order…… 9
parameters……. 22
why…… 2, 9
Documentation comments…….. 3
definition…… 36
Documentation conventions
classes……. 23
interfaces…… 25
packages……. 26

E

Encapsulation, and accessor methods.. 13
End of line comments……4, 20
English descriptors…… 2
Exception objects…….. 20

F

field…….See attribute
finalize() method, definition…… 36
FLOOT……. 29
Fragile base class problem…….. 18
Full Lifecycle Object-Oriented Testing. 29

G

Garbage collection, definition……. 37
Getter method…… 5
advantages and disadvantages…….. 18
and lazy initialization…….. 15
definition……. 37
for constants…….. 15
naming conventions……. 5, 15

H

HTML, definition…….. 37

I

Idioms…….. 31
Indenting…….……See paragraphing
Inheritance and coupling……. 17
Inline comments
definition……. 37
disadvantages…… 4
Interface…….. 25
definition…… 37
documentation convention…….. 25
naming conventions…….. 25
Invariant
definition……. 37
documentation of……..…12, 20

J

Java
and methods…….. 7
definition……. 37
doclets…….. 31
javadoc……. 4
javadoc tags…….….See tags
tags…….. 4
JDK, definition…….. 37

K

KISS 31

L

Lazy initialization……. 14
definition……. 37
documentation…….. 15
Local variable
coding standards…….. 20
declaring……. 20
definition…… 37
documentation of……9, 20
Long names……. 1
Loop Counters…… 20

M

Maintenance
and parenthesis…… 10
cost of…….. 1
Master test/QA plan……. 28
Method
accessors……..
coding standards…….. 5
definition……. 37
documentation…….. 7
getter…… 5
internal documentation…….. 8
setter…… 6
visibility…….. 6
Method signature……..See signature
Mixed case…….. 1
Modeling pattern, definition…….. 38
Mutator……See setter method

N

Name hiding…….. 12
  definition……. 38
Naming conventions
  accessor methods…….. 13
  accessors…….. 5
  attributes…… 11
  classes…….. 23
  components……. 11
  constants……. 11
  exception objects…… 20
  for parameters…….. 22
  getter methods…….. 5, 13
  interfaces…… 25
  local variables…… 20
  loop counters…….. 20
  methods…….. 5
  notifyAll…….. 18
  overview…….. 1
  packages……. 26
  setter methods……5, 13
  source code…….. 26
  streams…… 20

O

Optimization
  and portability…….. 27
  factors…….. 28
  leave to end……. 28
Order of operations…… 10
Overload, definition…….. 38
Override, definition…….. 38

P

Package…… 26
  definition……. 38
  documentation conventions…… 26
  naming conventions……. 26
Paragraphing……. 9
  definition……. 38
Parameter
  definition……. 38
  documentation conventions…… 22
  naming convention…… 22
Pattern, definition……. 38
Plan, master test 28
Portability and optimization…….. 27
Postcondition
  definition……. 38
  documentation of…….. 8
Precondition
  definition……. 38
  documentation of…….. 8
Prime Directive…….. 1
Private visibility…… 6
Process pattern, definition…….. 38
Process Patterns Resource page……. 31
Processing order, documentation of….. 10
Property……See attribute
Protected visibility…… 6
Public interface of a class…… 24
Public visibility……. 6
Punch cards…… 10

Q

Quality assurance……. 28
  definition (QA)…….. 38

S

Setter method…… 16
  advantages and disadvantages……. 18
….definition…… 38
Short forms…… 2, 32
Signature, definition……. 38
Single-line comments…….. 3
  definition……. 38
  usage…… 8
Source code
  coding standards……... 26
  definition……. 39
Static initializers…….. 19
Streams…… 20

T

Tags…….. 4
  definition……. 39
Test harnesses…… 28
Test in the Large……... 29
Test in the Small……... 29
Testing……. 29
  class testing, definition…… 36
  method testing, definition……... 38
Thirty-second rule……. 10
Tips
  access to attributes……... 13
  beware endline comments……... 4
  define the public interface first…….. 25
  define your priorities…… 28
  document closing braces…….. 9
  object-oriented testing……. 29
  use interfaces for parameter types… 22
Traceability……. 29

U

UML, definition……. 39
Underscores, use of……... 2
Users, working closely with…… 31

V

Visibility
definition……. 39
of accessor methods…….. 18
of attributes…… 12
of methods…….. 6

W

White space……. 9
definition……. 39
Why, documentation of……. 3
Widget……....See component