AmbySoft Inc.
Java Coding Standards
v17.01a

Scott W. Ambler
Senior Object Consultant

**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: July 10, 1997

Copyright 1997 AmbySoft Inc.

This page left unintentionally blank.
(yuk yuk yuk)

Copyright 1997 AmbySoft Inc.

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 enhanceability.

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 Me Improve These Standards

Because I welcome your input and feedback, please feel free to email me at **ambler@hookup.net** with your comments and suggestions. Let's work together and learn from one another.

Acknowledgments

The following people have provided valuable input into the development of these standards, and I would like to recognize them for it.

Stephan Marceau	IBM Canada
Eva Greff	ScotiaBank, Canada
Graham Wright	MDi Systems Ltd., UK

Scott W. Ambler

Copyright 1997 AmbySoft Inc.

May 1997

Copyright 1997 AmbySoft Inc.

This page also left unintentionally blank.
(although now it isn't quite as funny)

Copyright 1997 AmbySoft Inc.

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
1.4.1 The Three Types of Java Comments 3
1.4.2 A Quick Overview of javadoc 4

2. STANDARDS FOR METHODS 5

2.1 NAMING METHODS 5
2.1.1 Naming Accessor Methods 5
2.1.1.1 Getters 5
2.1.1.2 Setters 6
2.1.1.3 Constructors 6
2.2 METHOD VISIBILITY 7
2.3 DOCUMENTING METHODS 7
2.3.1 The Method Header 7
2.3.2 Internal Documentation 9
2.4 TECHNIQUES FOR WRITING CLEAN CODE 10
2.4.1 Document Your Code 10
2.4.2 Paragraph Your Code 10
2.4.3 Use Whitespace in Your Code 11
2.4.4 Follow The Thirty-Second Rule 11
2.4.5 Write Short, Single Command Lines 11
2.4.6 Specify the Order of Operations 11

3. STANDARDS FOR ATTRIBUTES (FIELDS/PROPERTIES) 12

3.1 NAMING ATTRIBUTES 12
3.1.1 Naming Components (Widgets) 12
3.1.1.1 Alternative for Naming Components – Hungarian Notation 12
3.1.1.2 Alternative for Naming Components – Postfix–Hungarian Notation 13
3.1.2 Naming Constants 13
3.1.3 Naming Collections 13
3.2 ATTRIBUTE VISIBILITY 14
3.2.1 Don't "Hide" Names 14
3.3 DOCUMENTING AN ATTRIBUTE 14
3.4 THE USE OF ACCESSOR METHODS 15
3.4.1 Naming Accessors 15
3.4.2 Advanced Techniques for Accessors 16
3.4.2.1 Lazy Initialization 16
3.4.2.2 Getters for Constants 17
3.4.2.3 Accessors for Collections 19
3.4.3 Visibility of Accessors 19
3.4.4 Why Use Accessors? 19
3.4.5 Why Shouldn't You Use Accessors? 20
3.5 ALWAYS INITIALIZE STATIC ATTRIBUTES 20

4. STANDARDS FOR LOCAL VARIABLES 22

4.1 NAMING LOCAL VARIABLES 22
4.1.1 Naming Streams 22
4.1.2 Naming Loop Counters 22

Copyright 1997 AmbySoft Inc.

4.1.3 Naming Exception Objects *22*
4.1.4 Bad Ideas for Naming Local Variables *23*
4.2 DECLARING AND DOCUMENTING LOCAL VARIABLES 23

5. STANDARDS FOR PARAMETERS (ARGUMENTS) TO METHODS 24

5.1 NAMING PARAMETERS 24
5.2 DOCUMENTING PARAMETERS 24

6. STANDARDS FOR CLASSES, INTERFACES, PACKAGES, AND COMPILATIONS UNITS ..26

6.1 STANDARDS FOR CLASSES 26
6.1.1 Naming Classes *26*
6.1.2 Documenting a Class *26*
6.1.3 Class Declarations *27*
6.1.4 Minimize the Public and Protected Interface *28*
6.2 STANDARDS FOR INTERFACES 28
6.2.1 Naming Interfaces *28*
6.2.2 Documenting Interfaces *29*
6.3 STANDARDS FOR PACKAGES 29
6.3.1 Naming Packages *29*
6.3.2 Documenting a Package *30*
6.4 STANDARDS FOR COMPILATION UNITS 30
6.4.1 Naming a Compilation Unit *30*
6.4.2 Documenting a Compilation Unit *30*

7. MISCELLANEOUS STANDARDS 32

7.1 REUSE 32
7.2 IMPORTING CLASSES 32

8. THE SECRETS OF SUCCESS 33

8.1 USING THESE STANDARDS EFFECTIVELY 33
8.2 OTHER FACTORS THAT LEAD TO SUCCESSFUL CODE 33

9. PROPOSED JAVADOC TAGS FOR METHODS 35

10. SUMMARY 36

10.1 JAVA NAMING CONVENTIONS 37
10.2 JAVA DOCUMENTATION CONVENTIONS 39
10.2.1 Java Comment Types *39*
10.2.2 What To Document *40*
10.3 JAVA CODING CONVENTIONS (GENERAL) 41

11. GLOSSARY 42

12. REFERENCES AND SUGGESTED READING 45

12.1 THE OBJECT PRIMER 47
12.2 BUILDING OBJECT APPLICATIONS THAT WORK 48

13. ABOUT THE AUTHOR 49

14. INDEX 50

Copyright 1997 AmbySoft Inc.

1. General Concepts

I'd like to start this white paper with a discussion of some general concepts that I feel are important for coding standards. I 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.

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's 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. Furthermore, unless your project is very small, you won't be the only one working on it. 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 don't know any better, will often fight having to follow standards. They claim they can code faster if they do it their own way. Pure hogwash. They MIGHT be able to get code out the door faster, but I doubt it. Cowboy programmers get hung up during testing when several difficult–to–find bugs crop up, and when their code needs to be enhanced it often leads to a major rewrite by them because they're the only ones who understand their code. Is this the way that you want to operate? I certainly don't.

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 don't apply. This leads me to introduce what I consider 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, understand when to apply them, and just as importantly when not to apply them.

1.3 What Makes Up a Good Name

We'll be discussing naming conventions 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/…** 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 don't provide any indication of what they represent and result in code that is difficult to understand, maintain, and enhance (Nagler, 1995; Ambler, 1997).

Copyright 1997 AmbySoft Inc.

2. **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).

3. **Use short forms sparingly, but if you do so then use them intelligently.** This means you should maintain a list of standard short forms, you should choose them wisely, and you should use them consistently. For example, if you want to use a short form for the word "number," then choose one of **nbr**, **no**, or **num**, document which one you chose (it doesn't really matter which one), and use only that one.

4. **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 (OK, I'm stretching it on this example) this name is simply too long and you should consider renaming it to something shorter, perhaps something like **Offering** (NPS, 1996).

5. **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).

6. **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). More importantly, underscores are annoying and difficult to type so I try to avoid their use whenever possible.

1.4 Good Documentation

We'll also be discussing documentation conventions, so let's discuss some of the basics first:

1. **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).

2. **If your program isn't worth documenting, it probably isn't worth running (Nagler, 1995).** What can I say, Nagler hit the nail on the head with this one.

3. **Avoid decoration, i.e. don't 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 gave them an outlet for their artistic urges, but frankly it was a major waste of time that added little value to the end product. You want to write clean code, not pretty code. Furthermore, because many of the fonts used to display and print your code are proportional, and many aren't, you can't line up your boxes properly anyway.

4. **Keep comments simple.** Some of the best comments I've ever seen are simple, point–form notes. You don't 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. Because documentation makes your code easier to understand you are able to take advantage of this fact while you are developing it. The way I look at

Copyright 1997 AmbySoft Inc.

it, if you're going to invest the time writing documentation you should at least get something out of it (Ambler, 1997).

6. **Document why something is being done, not just what.** Fundamentally, I can always look at a piece of code and figure out what it does. For example, I can look at the code in Example 1 below and figure out 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? I don't know unless it's documented somewhere, either in the source code itself or in an external document (Ambler, 1997).

```
if ( grandTotal >= 1000.00)
{
        grandTotal = grandTotal * 0.95;
}
```
Example 1.1

1.4.1 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 mysuggested 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 attributes 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.

The important thing is that your organization should set a standard as to how C−style comments and single−line comments are to be used, and then to follow that standard consistently. Use one type to document business logic, I prefer using single−line comments for that because I can put the documentation

Copyright 1997 AmbySoft Inc.

on the same line as the code (this is called inlining), and use the other to document out old code, I prefer C−style comments for this because I can comment out several lines at once and because C−style looks very similar to documentation comments I rarely use them so as to avoid confusion.

1.4.2 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. I think that *javadoc* is a great utility, but at the time of this writing it does have it's 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 I feel are not sufficient for adequately documenting your code. I'll expand upon this statement in chapter 9. For now, I present a brief overview of the current *javadoc* tags in the chart below, and will refer you 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.
@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.
@see ClassName	Classes, Interfaces, Methods, Attributes	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, Attributes	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's documented for you already.

Copyright 1997 AmbySoft Inc.

2. Standards For Methods

I'm a firm believer in maximizing the productivity of systems professionals. Because I also recognize that an application spends the majority of its existence being maintained, not developed, I am very interested in anything that can help to make my code easier to maintain and to enhance, as well as to develop. 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.

2.1.1 Naming Accessor Methods

We will discuss accessors, methods that get and set the values of attributes (fields/properties) in greater detail in chapter 3. The naming conventions for accessors, however, are summarized below.

2.1.1.1 Getters

Getters are methods that return the value of an attribute. You should prefix the word 'get' to the name of the attribute, unless it's a boolean attribute and then you prefix 'is' to the name of the attribute instead of 'get.'

Examples:

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

Copyright 1997 AmbySoft Inc.

By following this naming convention you make it obvious that a method returns an attribute 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). The main disadvantage is that 'get' is superfluous, requiring extra typing.

2.1.1.2 Setters

Setters are methods that modify the values of an attribute. You should prefix the word 'set' to the name of the attribute, regardless of the attribute 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 an attribute 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.

2.1.1.3 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 adhered to.

Copyright 1997 AmbySoft Inc.

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. If method doesn't have to be public then make it protected, and if it doesn't have to be protected then make it private.

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 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.

2.3.1 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, how they will be used, and what type of class they are an instance of. This information is needed so that other programmers know what information to pass to a method. The *javadoc* @*param* tag, discussed in section 1.4.2, 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 1.4.2, 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.

Copyright 1997 AmbySoft Inc.

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 1.4.2, is used for this.

6. **Visibility decisions.** If you feel that you 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 don't 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 method 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 it 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 method 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, 1997), 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 at best it's 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 an attribute, 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 it's synchronization you should also document why.

This 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. In chapter 9 I propose several new documentation tags for *javadoc* to support the factors listed above.

Copyright 1997 AmbySoft Inc.

2.3.2 Internal Documentation

In addition to the method documentation, you also need to include to include comments within your methods 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 1.4.1, you should seriously consider choosing one style of comments for documenting the business logic of your code and one for commenting out unneeded code. My 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 which follow at the end of a line of code. I use C−style comments to document out lines of unneeded code because I can easily take out several lines with only one comment. Furthermore, because C−style comments look so much like documentation comments I feel that their use can be confusing, taking away from the understandability of my 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's 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'll discuss this in greater detail in chapter 4, each local variable defined in a method should be declared on it's 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 don't have the time, then you must document thoroughly any complex code in a method. My 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, 1997). 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.

Copyright 1997 AmbySoft Inc.

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's 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. I have found that by marking the ending braces with an inline comment such as **//end if**, **//end for**, **//end switch**, … makes my code easier to understand.

Given the choice, however, I would rather use a more sophisticated editor.

2.4 Techniques for Writing Clean Code

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

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

2.4.1 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.

2.4.2 Paragraph 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 of within a block should be uniformly indented one unit[1]. 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 choose an indentation style and stick to it. My advice is to use the same indentation style that your Java development environment uses for the code that it generates.

[1] There seems to be a debate raging about what a unit is, but the only unit that makes sense to me is a horizontal tab. It requires the least amount of typing, one entire keystroke, while at the same time providing enough of an indent to be noticeable. I have always found that using spaces is problematic, some people use two, some three, some four, and so on. Tabs are much easier. I've been informed that some editors convert tabs to spaces (yuck!) and that others don't support tabs at all. My only response is to invest in a decent code editor.

Copyright 1997 AmbySoft Inc.

2.4.3 Use Whitespace in Your Code

A few blank lines, called whitespace, 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, 1997). 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 whitespace is very difficult to read and to understand.

2.4.4 Follow The Thirty−Second Rule

I've always believed that 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's probably too long.

2.4.5 Write Short, Single Command Lines

Your code should do one thing per line (Vision, 1996; Ambler, 1997). 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, but considering it's been over fifteen years since I've even seen a punch card I think we can safely rethink this approach to writing code. Whenever you attempt to do more than one thing on a single line of code you make it harder to understand. Why do this? We want to make our code 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 (assuming of course you aren't using punch cards).

Furthermore, you should write code that remains visible on the screen (Vision, 1996). My 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.

2.4.6 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, 1997). If I have to know the order of operations for a language to understand your source code then something is seriously wrong. This is mostly an issue for logical comparisons where you AND and OR several other comparisons together. Note that if you use short, single command lines as suggested above then this really shouldn't crop up as an issue.

Copyright 1997 AmbySoft Inc.

3. Standards for Attributes (Fields/Properties)

Throughout this white paper I will use the term *attribute* to refer to a field, which the Beans Development Kit (BDK) calls a property (DeSoto, 1997). An attribute is a piece of data that describes an object or class. Attributes may be base data type like a string or a float, or may be an object such as a customer or bank account.

3.1 Naming Attributes

You should use a full English descriptor to name your attributes (Gosling, Joy, Steele, 1996; Ambler 1997) to make it obvious what the attribute represents. Attributes 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

3.1.1 Naming Components (Widgets)

For names of components (interface widgets) you should use a full English descriptor postfixed by the widget type[2]. 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**, …).

Examples:

okButton
customerList
fileMenu
newFileMenuItem

3.1.1.1 Alternative for Naming Components – Hungarian Notation

The "Hungarian Notation" (McConnell, 1993) is based on the principle that an attribute should be named using the following approach: xEeeeeeEeeeee where x indicates the component type and EeeeeEeeeee is the full English descriptor.

Examples:

pbOk
lbCustomer
mFile
miNewFile

[2] This is my own standard and not one promoted by Sun.

Copyright 1997 AmbySoft Inc.

The main advantage is that this is an industry standard common for C++ code so many people already follow it. Furthermore, developers can quickly judge from the name of the variable its type and how it is used. The main disadvantages are that the prefix notation becomes cumbersome when you have a lot of the same type of widget and you break from the full English descriptor naming convention.

3.1.1.2 Alternative for Naming Components – Postfix–Hungarian Notation

Basically a combination of the other two alternatives, resulting in names such as **okPb**, **customerLb**, **fileM**, and **newFileMi**. The main advantage is that the name of the component indicates the widget type and that widgets of the same type aren't grouped together in an alphabetical list. The main disadvantage is that you still aren't using a full English description, making the standard harder to remember because it deviates from the norm.

Tip – Set Component Name Standards

Whatever convention you choose, you'll want to create a list of "official" widget names. For example, when naming buttons do you use **Button** or **PushButton**, **b** or **pb**? Create a list and make it available to every Java developer in your organization.

3.1.2 Naming Constants

In Java, constants, values that do not change, at typically implemented as *static final* attributes 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. In section 3.4.2.3 we will see 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.

3.1.3 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

Copyright 1997 AmbySoft Inc.

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

3.2 Attribute Visibility

The Vision team (1996) suggest that attributes not be declared *public* for reasons of encapsulation, but I would go further to state that all attributes should be declared *private*. When attributes 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. Attributes should never be accessed directly, instead accessor methods (see below) should be used.

Visibility	Description	Proper Usage
public	A public attribute can be accessed by any other method in any other object or class.	Don't make attributes public.
protected	A protected attribute can be accessed by any method in the class in which it is declared or by any methods defined in subclasses of that class.	Don't make attributes protected.
private	A private attribute can only be accessed by methods in the class in which it is declared, but not in the subclasses.	All attributes should be private and be accessed by getter and setter methods (accessors).

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

3.2.1 Don't "Hide" Names

Name hiding refers to the practice of naming a local variable, argument, or attribute the same (or similar) as that of another one of greater scope. For example, if you have an attribute called **firstName** don't create a local variable or parameter called **firstName**, or anything close to it like **firstNames** or **fistName** (hey, maybe some people name their fists, personally I refer to mine as "left" and "right" <grin>). 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 an Attribute

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

1. **It's description.** You need to describe an attribute so that people know how to use it.

2. **The type of an attribute.** Other developers need to know whether an attribute is a number or a string. Whether it's another object, such as a customer or bank account, or if it's a boolean. By documenting the type of an attribute you provide valuable information to other developers as to how it is to be used.

Copyright 1997 AmbySoft Inc.

3. **Document all applicable invariants.** Invariants of an attribute are the conditions that are always true about it. For example, an invariant about the attribute **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 attribute based on the month and the year). By documenting the restrictions on the value of an attribute you help to define important business rules, making it easier to understand how your code works (or at least should work).

4. **Examples.** For attributes 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's worth a thousand words.

5. **Concurrency issues.** Concurrency is a new and complex concept for many developers, actually, at best it's 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.

6. **Visibility decisions.** If you've declared an attribute to be anything but private then you should document why you have done so. Attribute visibility is discussed in section 3.2 above, and the use of accessors methods to support encapsulation is covered in section 3.4 below. The bottom line is that you'd 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 attributes is achieved by the appropriate use of *accessor methods*, methods that provide the functionality to either update an attribute or to access its value. Accessor methods come in two flavors: *setters* 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, this is no longer true. Accessors help to hide the implementation details of your class. By having at most two control points from which a variable is accessed, one setter and one getter, you are able to increase the maintainability of your classes by minimizing the points at which changes need to be made.

One of the most important standards that your organization can enforce is the use of accessors. Some developers don't want to use accessor methods because they don't 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 attribute). The bottom line is that the increased maintainability and extensibility from using accessors more than justifies their use.

3.4.1 Naming Accessors

Getter methods should be given the name 'get' + attribute name, unless the attribute represents a boolean (true or false) and then the getter is given the name 'is' + attribute name. Setter methods should be given the name 'set' + attribute name, regardless of the attribute type (Gosling, Joy & Steele, 1996; DeSoto, 1997). Note that the attribute 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:

Attribute	Type	Getter name	Setter name

Copyright 1997 AmbySoft Inc.

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

3.4.2 Advanced Techniques for Accessors

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

- Initialize the values of attributes
- Access constant values
- Access collections

3.4.2.1 Lazy Initialization

All 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 methods 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 attributes are initialized by their getter methods, as shown below[3]. Notice the method 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 attributes that are actually other objects stored in the database. For example, when you create a new inventory item you don't 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

[3] Note how a setter method is used within the getter method.

Copyright 1997 AmbySoft Inc.

when and if you need it. This is approach is advantageous for objects that have attributes 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 method 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 attributes are used in your code, improving both its maintainability and extensibility.

3.4.2.2 Getters for Constants

The common Java wisdom, perhaps wisdom is the wrong word, is to implement constant values as static final attributes. This approach makes sense for "constants" that are guaranteed to be stable. For example the class **Boolean** implements two *static final* attributes 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[4].

However, many so–called business constants change over time because they constant value 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 attribute 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?[5] 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 attributes 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 attribute called **MINIMUM_BALANCE** because we can implement the various business rules in this method and subclass it appropriately for various kinds of accounts.

[4] I'm assuming that all cultures have a 7–day week, something that I don't know for sure. I've been involved with the development of enough international applications to know that I really need to verify this assumption. You really learn a lot on projects involving localization issues, I highly suggest getting on one.

[5] Hey, I'm Canadian. This could happen.

Copyright 1997 AmbySoft Inc.

```
/**
        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 int 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)
   {
      int newAccountNumber;   // The rightmost 6 digits of the full account number

      newAccountNumber = newNumber % 1000000;
      setBranchAccountNumber(newAccountNumber);
      setBranchNumber(newNumber –newAccountNumber);
   }
```

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's because we used 1,000,000 instead of 100,000 to extract the attribute **branchAccountNumber**. Had we used a single source for this value, the constant getter **getAccountNumberDivisor()** as we see in below, our code would have been more consistent and would have worked.

```
/**
        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 (100000);
   }

/**
        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 (100000);
   }
```

Copyright 1997 AmbySoft Inc.

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.

3.4.2.3 Accessors for Collections

The main purpose of accessors is to encapsulate the access to attributes so as to reduce the coupling within your code. Collections, such as arrays and vectors, being more complex than single value attributes 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. The approach that I use is to add the following accessor methods where appropriate for an attribute which is a collection:

Method type	Naming convention	Example
Getter for the collection	**getCollection()**[6]	**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()**

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.

3.4.3 Visibility of Accessors

Always strive to make them protected, so that only subclasses can access the attributes. Only when an 'outside class' needs to access an attribute 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 an attribute representing a collection of **OrderItem** instances, and a second attribute called **orderTotal** which is the total of the entire order. The **orderTotal** is a convenience attribute 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.

3.4.4 Why Use Accessors?

I think that 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

[6] Remember, 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.

Copyright 1997 AmbySoft Inc.

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

2. **Obtaining the values of attributes**. You have complete control over how attributes 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 attributes**. The use of lazy initialization ensures that attributes 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 attributes only through their corresponding accessor methods, it makes it possible to change the implementation of attributes in the superclass without affecting any of its subclasses, effectively reducing coupling between them.

6. **Encapsulating changes to attributes**. If the business rules pertaining to one or more attributes 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 attribute. 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 attributes, the use of accessors to always access attributes means that you can give local variables any name you want – You don't have to worry about hiding attribute names because you never access them directly anyway.

3.4.5 Why 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 attributes in combination must be consistent, and that it isn't wise to provide access to attributes singly. He's right, so don't! I think that Lea has missed the point that you don't need to make all accessor methods public. When you are in the situation that the values of some attributes 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 don't have to make all of your accessors public.

3.5 Always Initialize Static Attributes

Doug Lea (1996) validly points out that you must ensure that static attributes, also known as class attributes, be given valid values because you can't assume that instances of a class will be created before a

Copyright 1997 AmbySoft Inc.

static attribute 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 attributes – With accessor methods you can always use lazy initialization to guarantee that the value of an attribute is set. The use of accessor methods to encapsulate attributes gives you complete control over how the are used, while reducing coupling within your code. A win–win situation.

Copyright 1997 AmbySoft Inc.

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 method. 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 attributes, in other words used 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

4.1.1 Naming Streams

When there is a single input and/or output stream being opened, used, and then closed within a method 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.**

A common alternative to this naming convention is to use the names **inputStream**, **outputStream**, and **ioStream** instead of **in**, **out**, and **inOut** respectively. To tell you the truth I like this alternative better, but the fact remains that the names **in** and **out** are what Sun suggests, so that's what you should probably stick with.

4.1.2 Naming Loop Counters

Because loop counters are a very common use for local variables, and because it was acceptable in C/C++, in Java programming the use of **i**, **j**, or **k**, is acceptable for loop counters (Gosling, Joy, Steele, 1996; Sandvik, 1996). If you use these names for loop counters, use them consistently.

A common alternative is to use names like **loopCounter** or simply counter, but the problem with this approach is that you often find names like **counter1** and **counter2** in methods that require more than one counter. The bottom line is that **i**, **j**, **k** work as counters, they're quick to type in, and they're generally accepted.

4.1.3 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).

Copyright 1997 AmbySoft Inc.

4.1.4 Bad Ideas for Naming Local Variables

Unfortunately a few "common" short names have been approved for local variables, but frankly I think they're inappropriate. In the chart below I have summarized the naming conventions put forth by Sun for other types of common objects/variables used in Java (Gosling, Joy, Steele, 1996). I'm sorry, but it is my opinion that these conventions have crossed the hacking line are actually take away from the readability of your source code. Use these conventions if you want to, but I don't recommend them.

Variable Type	Suggested Naming Convention
offset	**off**
length	**len**
byte	**b**
char	**c**
double	**d**
float	**f**
long	**l**
Object	**o**
String	**s**
Arbitrary value	**v**

4.2 Declaring and Documenting Local Variables

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

1. **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).

2. **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. You should document what a local variable is used for and where appropriate why it is used, making your code easier to understand.

3. **Declare local variables immediately before their use.** By declaring local variables where they are first needed other programmers do not need to scroll to the top of the method to find out what a local variable is used for. Furthermore, your code may be more efficient because if that code is never reached the variable will not need to be allocated (Vision, 1996). The main disadvantage of this approach is that your declarations are dispersed throughout each of your methods, making it difficult to find all declarations in large methods.

4. **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 from earlier in the code. 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.

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 2.4.5, 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.

Copyright 1997 AmbySoft Inc.

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 I will use the term *parameter* to refer to a method argument.

5.1 Naming Parameters

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

Examples:

customer
inventoryItem
photonTorpedo
in
e

A viable alternative, 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 attributes, and avoids the name hiding problem. This is my preferred approach.

Examples:

aCustomer
anInventoryItem
aPhotonTorpedo
anInputStream
anException

Another alternative to naming variables, which I don't recommend at all, is to give them a name based on their type. This is a bad idea for several reasons: First, the type is already indicated in the definition of the method. Second, a parameter name like **aString** tells you much less information than a name like **accountNumber** or **firstName**.

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. **The type of the parameter.** Other developers need to know the type of a parameter, for example a string or a customer object.

2. **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.

Copyright 1997 AmbySoft Inc.

3. **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.

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

Copyright 1997 AmbySoft Inc.

6. Standards for Classes, Interfaces, Packages, and Compilations Units

This chapter concentrates on standards and guidelines for classes, interfaces, packages, and compilation units. A class is a template from which objects are instantiated (created). Classes contain the declaration of attributes and methods. Interfaces are the definition of a common signature, including both methods and attributes, which a class that implements an interface must support. A package is a collection of related classes. Finally, a compilation unit is a source code file in which classes and interfaces are declared. Because Java allows compilation units to be stored in a database an individual compilation unit 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

6.1.1 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, 1997).

Examples:

Customer
Employee
Order
OrderItem
FileStream
String

6.1.2 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. I also make it a habit to document any good things to know about a class, for example is it part of a pattern or are there any interesting limitations to using it (Ambler, 1997).

2. **Known bugs[7].** 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

[7] Yes, it's better to fix bugs. However, sometimes you don't 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 that when passed a negative number doesn't work properly, but it does work properly for positive numbers. Your application only

Copyright 1997 AmbySoft Inc.

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.

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 who 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 it's 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's 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.

6.1.3 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 attributes
protected methods
protected attributes
private methods
private attributes

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 we have a standard to declare all attributes as private, the declaration order really boils down to:

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

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.

---

passes it positive numbers, so you can live with the bug but decide to be polite and document that the problem exists.

Copyright 1997 AmbySoft Inc.

6.1.4 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're 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 method. 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.

6.2 Standards for Interfaces

The standards that are important for interfaces are based on:
- Naming conventions
- Documentation conventions

6.2.1 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).

Alternatives:

1. **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. I like this potential naming convention for the simple fact 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, and I don't see them ever changing. Therefore I chose the defacto standard described above.

2. **Postfix 'Ifc' onto the interface name.** Lea (1996) suggests appending 'Ifc' to the end of an interface name, resulting in names like **SingletonIfc** or **RunnableIfc**, whenever the name of an interface is similar to that of a class (Lea, 1996). I like the general idea, although I would be tempted to prefix

Copyright 1997 AmbySoft Inc.

the name with the full word 'Interface.' This suggestion suffers from the same problem as the one above.

6.2.2 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 don't 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 developer new to object–orientation, I suspect 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 how 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

6.3.1 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. **Local package names begin with an identifier that is not all upper case.** Local packages are ones that are used internally within your organization and that will not be distributed to other organizations. Examples of these package names include **persistence.mapping.relational** and **interface.screens**.

4. **Global package names begin with the reversed Internet domain name for your organization with the first identifier all in upper case.** A package that is to be distributed to multiple organizations should include the name of the originating organization's domain name, with the top–

Copyright 1997 AmbySoft Inc.

level domain type capitalized. For example, to distribute the previous packages, I would name them **COM.AmbySoft.www.persistence.mapping.relational** and **COM.AmbySoft.www.interface.screens** (note how the first identifier, **COM**, is in uppercase which signifies that the package is global).

6.3.2 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 don't have to worry about accidentally overwriting one package documentation file with another. I like Lea's idea, in general, but my experience has shown that similarly named files get overwritten often enough to modify his approach slightly.

6.4 Standards for Compilation Units

The standards and guidelines for compilation units are based on:

- Naming conventions
- Documentation conventions

6.4.1 Naming a Compilation Unit

A compilation unit, in this case a source code file, 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.

Examples:

Customer.java
Singleton.java
SavingsAccount.java

6.4.2 Documenting a Compilation Unit

Although you should strive to have only one class or interface declaration per file, it sometimes makes sense to define several classes (or interfaces) in the same file. My general rule of thumb is that if the sole purpose of class B is to encapsulate functionality that is needed only by class A then it makes sense that

Copyright 1997 AmbySoft Inc.

class B appear in the same source code file as class A[8]. As a result, I've separated the following documentation conventions that apply to a source code file, and not specifically to a class:

1. **For files with several classes, list each class.** If a file contains more than one class you should provide a list of the classes and a brief description for each (Lea, 1996).

2. **The file name and/or identifying information.** The name of the file should be included at the top of it (Lea, 1996). The advantage is that if the code is printed you know what the source file for the code is.

3. **Copyright information.** If applicable you should indicate any copyright information for the file (Lea, 1996). It is common to indicate the year of the copyright and the name of the individual/organization that holds the copyright. Note that the code author may not be the holder of the copyright.

[8] The concept of an "inner" class was introduced in JDK 1.1, which would be one option for implementing class B.

Copyright 1997 AmbySoft Inc.

7. Miscellaneous Standards

In this chapter I want to share with you several standards/guidelines that are important, but that are general enough that they need their own chapter.

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 a variety of sources, either a third–party development company that specializes in Java libraries or another division or project team within your organization.

Scott's Soapbox – 100% Pure is 100% Dead On

In my opinion the 100% Pure effort from Sun is exactly what Java needs. In my second book, *Building Object Applications That Work* (Ambler, 1997), I had a few harsh words about the portability of Java. There are two portability issues with Java: Source code portability and bytecode portability. At the time of this writing, May 1997, I think that developers porting from JDK 1.0 to JDK 1.1 now have a better appreciation for what I was talking about. 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 whom 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.

7.2 Importing Classes

The import statement allows the use of wildcards when indicating the names of classes. For example, the statement

import java.awt.*;

brings in all of the classes in the package **java.awt** at once. Actually, that's not completely true. What really happens is that every class that you use from the **java.awt** package will be brought into your code when it is compiled, classes that you don't use won't be. Although this sounds like a good feature, it reduces the readability of your code. A better approach is to fully qualify the name of the classes that your code uses (Laffra, 1997; Vision, 1996). A better way to import classes is shown in the example below:

```
import java.awt.Color;
import java.awt.Button;
import java.awt.Container;
```

Copyright 1997 AmbySoft Inc.

8. The Secrets of Success

The good news is that the reason why I wrote this white paper is because I want 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, don't declare each local variable on its own line just because I told you to, 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. It's been years since I've had to do an "all–nighter" writing code, in most part because I make it a point to use tools and techniques that make me a productive developer. I believe in following standards because it's been my experience that intelligent standards applied appropriately lead to significant increases in my productivity as a developer.

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 attributes 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). I've seen 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's stupid. If you write clean code right from the beginning you can benefit from it while you are creating it. That's 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 don't 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, 1997) 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 nobody else can figure it out, then it isn't any good. Use naming conventions. Document your code. Paragraph it.

Copyright 1997 AmbySoft Inc.

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're 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've always found that developing in small steps, writing a few methods, 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's 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're testing your code and you find a bug you almost always find that 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. **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 don't understand the needs of your users, and the only way that you can understand their needs is if you work closely with them.

5. **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 a jerk? Writing code that needs to be rewritten later is nothing to be proud of, so follow the KISS rule: Keep it simple, stupid.

Copyright 1997 AmbySoft Inc.

9. Proposed javadoc Tags for Methods

From the discussions about documentation in this white paper it should be clear that *javadoc* needs a few more tags. While I grant that it's important to keep *javadoc* simple, at the same time it also needs to be sufficient for the task at hand. As a result, I wish to propose the following new tags that I hope will be supported in a future version of *javadoc*[9].

Proposed Tag	Used for	Purpose
@bug description	Classes, Methods	Describes a known bug with the class or method. One tag per bug should be used.
@concurrency description	Classes, Methods, Attributes	Describes the concurrency strategy/approach taken by the class/method/attribute. The execution context for a method should be described at this point.
@copyright year description	Classes	Indicates that a class is copyrighted, the year that it was copyrighted in, and any descriptive information such as the name of the individual/organization that holds the copyright.
@example description	Classes, Methods, Attributes	Provides one or more examples of how to use a given class, method, or attribute. This helps developers to quickly understand how to use your classes.
@fyi description	Classes, Interfaces Methods, Attributes	Provides information about design decisions that you have made and/or good things to know about parts of your code.
@history description	Classes, Methods	Describes how a class/method has been modified over time. For each change, the description should include: who made it, when it was made, what was done, why it was done, and a reference to the change request and/or user requirement that resulted in it.
@modifies no	Methods	Indicates that a method does not modify an object
@modifies yes description	Methods	Indicates that a method modifies an object and describes how it does so.
@postcondition description	Methods	Describes a postcondition that is true after a method has been invoked. Use one tag per postcondition.
@precondition description	Methods	Describes a precondition that must be true before the method can be safely invoked. Use one tag per precondition.
@reference description	Classes, Interfaces, Methods, Attributes	Describes a reference to an external document that documents pertinent business rules or information that is relevant to the source code being documented.
@values	Attributes	Describes the possible values of an attribute, including ranges and/or distinct values.

[9] In June of 1997 I will submit the full text of this section to SunSoft as a proposed improvement to *javadoc*.

Copyright 1997 AmbySoft Inc.

10. Summary

In this white paper we discussed many standards and guidelines for Java developers. Because this white paper is reasonably large I have summarized them here for your convenience. I highly suggest reprinting the pages of this chapter and pinning them on the wall of your workspace so that they are at your fingertips.

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.

Copyright 1997 AmbySoft Inc.

10.1 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 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, possibly prefixing the name with 'a' or 'an.' The important thing is to choose one approach and stick to it.	**customer**, **account**, – or – **aCustomer**, **anAccount**
Attributes/ fields/ properties	Use a full English description of the attribute, 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 attributes described above then you simply give it the name of the attribute.	**isPersistent()**, **isString()**, **isCharacter()**
Classes	Use a full English description, with the first letters of all words capitalized.	**Customer**, **SavingsAccount**
Compilation unit 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 which 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 attributes (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**

Copyright 1997 AmbySoft Inc.

Java Naming Conventions Continued

Item	Naming Convention	Example
Getter methods	Prefix the name of the attribute 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. It is customary to postfix the name with either 'able,' 'ible,' or 'er' but this is not required.	**Runnable**, **Contactable**, **Prompter**, **Singleton**
Local variables	Use full English descriptions with the first letter in lower case but do not hide existing attributes/fields. For example, if you have an attribute named 'firstName' don't have a local variable called 'firstName.'	**grandTotal**, **customer**, **newAccount**
Loop counters	It is generally accepted to use the letters **i**, **j**, or **k**, or the name '**counter**.'	**i**, **j**, **k**, **counter**
Packages	Use full English descriptions, using mixed case with the first letter of each word in uppercase, everything else in lower case. For global packages, reverse the name of your Internet domain, put the first identifier completely in uppercase, and concatenate to this the package name.	**java.awt**, **COM.AmbySoft.www.persistence.mapping**
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 attribute being accessed with 'set'.	**setFirstName()**, **setLastName()**, **setWarpSpeed()**

Although I don't agree with the following conventions, Sun suggests that for local variables of the given types you can give them short names. A much better convention is to use a full English descriptor – Don't be lazy.

Variable Type	Suggested Naming Convention
offset	**off**
length	**len**
byte	**b**
char	**c**
double	**d**
float	**f**
long	**l**
Object	**o**
String	**s**
Arbitrary value	**v**

Copyright 1997 AmbySoft Inc.

10.2 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. don't use banner–like comments
- Keep comments simple
- Write the documentation before you write the code
- Document why something is being done, not just what

10.2.1 Java Comment Types

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

Comment Type	Usage	Example
Documentation	Use documentation comments immediately before declarations of interfaces, classes, methods, and attributes 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.[10]	/* 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.

[10] This isn't actually a standard, it's more of a guideline. The important thing is that your organization should set a standard as to how C–style comments and single–line comments are to be used, and then to follow that standard consistently.

Copyright 1997 AmbySoft Inc.

10.2.2 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
Attributes/ fields/properties	It's description The type of an attribute 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
Compilation units	Each class/interface defined in the class, including a brief description The file name and/or identifying information Copyright information
Getter method	Document why lazy initialization was used, if applicable
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

Copyright 1997 AmbySoft Inc.

10.3 Java Coding Conventions (General)

There are many conventions and standards which are critical to the maintainability and enhanceability 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 attributes in the database Use accessors for obtaining and modifying all attributes Use accessors for 'constants" For collections, add methods to insert and remove items Whenever possible, make accessors protected, not public
Attributes	Attributes should always be declared private Don't directly access attributes, instead use accessor methods Don't use final static attributes (constants), instead use accessor methods Don't hide names Always initialize static attributes
Classes	Minimize the public and protected interfaces Declare the attributes and methods of a class in the following order: • constructors • **finalize()** • public methods • protected methods • private methods • private attributes
Local variables	Don't 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 whitespace, 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

Copyright 1997 AmbySoft Inc.

11. 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 an attribute. Also known as an access modifier. See getter and setter.

Argument – See parameter.

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

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 also referred to as 'curlies' (don't ask).

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

Compilation unit – 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.

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.

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 attributes should be written with documentation comments.

Copyright 1997 AmbySoft Inc.

Field – See attribute.

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 an attribute. A getter can be used to answer the value of a constant, which is often preferable to implementing the constant as a static attribute 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 attributes, 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 an attribute is initialized in its corresponding getter method the first time that it is needed. Lazy initialization is used when an attribute 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.

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.

Name hiding – This refers to the practice of using the same, or at least similar, name for an attribute/variable/argument as for one of higher scope. The most common abuse of name hiding is to

Copyright 1997 AmbySoft Inc.

name a local variable the same as an instance attribute. 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.

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.

Property – See attribute.

Setter – An accessor method that sets the value of an attribute.

Signature – The combination of 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.

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*.

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

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

Widget – See component.

Copyright 1997 AmbySoft Inc.

12. References and Suggested Reading

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

Ambler, S.W. (1997). *Building Object Applications That Work: A Step-By-Step Handbook for Developing Systems Using Object Technology*. New York: SIGS Books.

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.

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

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.

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

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.

Copyright 1997 AmbySoft Inc.

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

Copyright 1997 AmbySoft Inc.

12.1 The Object Primer

The Object Primer is a straightforward, easy to understand introduction to object–oriented analysis and design techniques. Object–orientation is the most important change to system development since the advent of structured methods. While OO is often used to develop complex systems, OO itself does not need to be complicated. This book is different than any other book ever written about object–orientation (OO) – It's written from the point of view of a real–world developer, somebody who has lived through the difficulty of learning this exciting new approach. Readers of *The Object Primer* have found it to be one of the easiest introductory books in OO development on the market today, many of whom have shared their comments and kudos with me.

The Strengths of The Object Primer:

- Relates new OO concepts back to your existing knowledge – You can see how everything makes sense
- Covers proven, leading–edge object–oriented (OO) techniques and concepts in everyday language
- Discusses in detail why we need object–orientation
- Presents leading–edge OO modeling techniques using real–world examples that you can actually recognize
    - CRC modeling
    - Use–case scenarios
    - Class/object modeling
- Is written by a developer for developers, so it's easy to understand
- Is THE book to read before you take your first course in OO development – Get more out of the learning experience by understanding the basics ahead of time
- Is short, only 248 pages – You don't have to spend your entire life reading it
- Includes a consistent, easy–to–follow notation that isn't too complex for novices
- Puts the OO development process into perspective, presenting both an iterative and a serial approach

To Order The Object Primer:

1. Go to your local bookstore (if it isn't on the shelf then tell them to order it in!)
2. Order it online from SIGS Books (http://www.sigs.com/books)
3. Call SIGS at: 1–800–871–SIGS
4. Order it online from Amazon (http://www.amazon.com)

For more information visit http://www.AmbySoft.com

Copyright 1997 AmbySoft Inc.

12.2 Building Object Applications That Work

Building Object Applications That Work starts exactly where *The Object Primer* left off –– At OO design. Like *The Object Primer,* it is written in everyday language that you can understand, covering leading–edge topics that are pertinent to real–world developers. While OO is often used to develop complex systems, OO itself does not need to be complicated. Like *The Object Primer,* this book is written from the point of view of a developer, somebody who has lived through the difficulty of learning this exciting new approach to creating applications.

Uses the

Unified Modeling Language

Building Object Applications That Work is about:

- Architecting your applications so that they're maintainable and extensible
- Analysis, so you know what you have to build
- Design techniques, so that you know how you're going to build your application
- Creating applications for stand–alone, client/server, and distributed environments
- Using both relational and object–oriented (OO) databases to make your objects persistent
- Collecting the right metrics to improve your development approach
- Applying OO patterns to improve the quality of your applications
- Testing approaches to ensure that your applications work
- User interface design, so your users can actually work with the systems that you build
- Coding applications in a way that makes them maintainable and extensible

Summarized Table of Contents:

Chapter 1: Where we've been – Object–Oriented Concepts and Techniques
Chapter 2: Bubbles and Lines – Useful Diagrams for Object–Oriented Analysis and Design
Chapter 3: Improving Your Design – A Class–Type Architecture
Chapter 4: Reusing Your Development Efforts – Object–Oriented Patterns
Chapter 5: Development in the 90s and Beyond – Distributed Object–Oriented Applications
Chapter 6: Measuring and Improving the Quality of Your Work – Object–Oriented Metrics
Chapter 7: Choosing an Object–Oriented Language – Comparing the Leading Languages
Chapter 8: Building Your Application – Effective Object–Oriented Construction Techniques
Chapter 9: Making Your Applications Usable – Object–Oriented User Interface Design
Chapter 10: Making Your Objects Persistent – Object–Orientation and Databases
Chapter 11: Integrating Legacy Code – Wrapping
Chapter 12: Making Sure Your Applications Work – Full–Life Cycle Object–Oriented Testing
Chapter 13: Where to Go From Here – Personal Success Strategies

To Order Building Object Applications That Work:

1. Go to your local bookstore (if it isn't on the shelf then tell them to order it in!) Note: Books in Print might not have the book listed until August 1997.
2. Order it online from SIGS Books (http://www.sigs.com/books)
3. Call SIGS at: 1–800–871–SIGS
4. Order it online from Amazon (http://www.amazon.com)

For more information visit http://www.AmbySoft.com

Copyright 1997 AmbySoft Inc.

13. About the Author

Scott W. Ambler is a Object Development Consultant living in Sharon, Ontario, a small village 60 km north of Toronto, Canada. He has worked with OO technology since 1990 in various roles: Business Architect, System Analyst, System Designer, Smalltalk programmer, Java programmer, and C++ programmer. He has also been active in education and training as both a formal trainer and as an object mentor.

Scott has a Master of Information Science and a Bachelor of Computer Science from the University of Toronto and is the author of the best−selling books*The Object Primer* and *Building Object Applications That Work*, both published by SIGS Books (http://www.sigs.com). Scott is a contributing editor with *Software Development* (http://www.sdmagazine.com), writes a column for *Computing Canada* (http://www.plesman.com), and has had feature articles appear in *Object Magazine* and *Client/Server Computing*.

He can be reached via e−mail at:

ambler@hookup.net

and you can visit his personal web site:

http://www.AmbySoft.com

Copyright 1997 AmbySoft Inc.

14. Index

@

@author tag 4
@concurrency tag (proposed) 35
@example tag (proposed) 35
@exception tag 4
  usage 8
@fyi tag (proposed) 35
@history tag (proposed) 35
@modifies tag (proposed) 35
@param tag 4
  usage 7
@postcondition tag (proposed) 35
@precondition tag (proposed) 35
@return tag 4
  usage 7
@see tag 4
@version tag 4

1

100% pure
  definition 42

A

Accessor method 15
  advantages and disadvantages 19
  constant getters 17
  definition 42
  for collections 19
  getters 5
  overhead of 15
  setter methods 6
  visibility 19
Argument ..... *See* parameter
Attribute
  accessors 15
  and lazy initialization 16
  definition 42
  documenting 14
  getter methods 15
  initialization 16
  name hiding 14
  naming conventions 12
  setter methods 15
  visibility 14
Author
  contacting 49

B

Banner comments 2
BDK 42
Block
  definition 42
Braces
  definition 42
  documentation of 10

C

Class 26
  definition 42
  documentation conventions 26
  naming conventions 26
Clean code 10
Code clarity 2
Code history 8
Code inspections 33
Coding standards
  classes 26
  compilation units 30
  for local variables 22
  for methods 5
  importance of 1
  interfaces 28
  packages 29
  parameters 24
  using them effectively 33
Collections
  accessor methods 19
Comments
  banner style 2
  in methods 7
  types of 3
Compilation unit
  coding standards 30
  definition 42
  documentation conventions 30
  naming conventions 30
Complex code 9
Component
  definition 42
  naming conventions 12
Concurrency
  and setter methods 20
  documentation of 8, 15, 27
  notifyAll 20
Constant getter 17

Copyright 1997 AmbySoft Inc.

definition....42
Constants
naming conventions....13
Constructor
definition....42
Containment
definition....42
Control structures....9
Coupling
and accessors....20
and inheritance....20
C−style comments....3
definition....42
usage....9

D

Design first....34
Destructor
definition....42
Develop in small steps....34
Documentation
clarity....2
code history....8
concurrency....8, 15
internal....9
of attributes....14
of closing braces....10
of complex code....9
of local variables....23
of methods....7
of processing order....9
overview....2
parameters....24
why....3, 7
Documentation comments....3
definition....42
Documentation conventions
classes....26
compilation units....30
interfaces....29
packages....30

E

Encapsulation
and accessor methods....20
English descriptors....1
Exception objects....22

F

Field....*See* attribute
finalize() method
definition....43

G

Garbage collection
definition....43
Getter method....15
advantages and disadvantages....19
and lazy initialization....16
definition....43
for constants....17
naming conventions....5, 15

H

HTML
definition....43
Hungarian notation
postfix....13
prefix....12

I

Indenting....*See* paragraphing
Inheritance
coupling....20
Inline comments
definition....43
Interface....28
definition....43
documentation convention....29
naming conventions....28
Invariant
definition....43
documentation of....27

J

Java
definition....43
javadoc....4
and methods....7
definition....43
proposed tags....35
tags....4
javadoc tags....*See* tags
JDK....43

K

KISS....34

L

Lazy initialization....16
definition....43
documentation....17
Local variable
coding standards....22
declaring....23
definition....43

Copyright 1997 AmbySoft Inc.

documentation of....9
documenting....23
Long names....2
Loop counters....22

M

Maintenance
and parenthesis....11
cost of....1
Method
accessors....15
coding standards....5
definition....43
documentation....7
getter....15
internal documentation....9
setter....15
visibility....7
Method signature....*See* signature
Mixed case....2

N

Name hiding....14
definition....43
Naming conventions
accessor methods....15
accessors....5
attributes....12
classes....26
compilation units....30
components....12
constants....13
exception objects....22
for parameters....24
getter methods....5, 15
hungarian notation....12
interfaces....28
local variables....22
loop counters....22
methods....5
overview....1
packages....29
setter methods....6, 15
streams....22
notifyAll....20

O

Order of operations....11
Overload
definition....44
Override
definition....44

P

Package....29
definition....44
documentation conventions....30
naming conventions....29
Paragraphing....10
definition....44
Parameter
definition....44
documentation conventions....24
documentation of....7
naming convention....24
Parameters....24
Postcondition
definition....44
documentation of....8
Precondition
definition....44
documentation of....8
Prime Directive....1
Processing order
documentation of....9
Property....*See* attribute
Punch cards....11

Q

Quality assurance....33

S

Scott Ambler
contacting....49
Setter method....15
advantages and disadvantages....19
definition....44
naming conventions....6, 15
Short forms....2, 37
Signature
definition....44
Single−line comments....3
definition....44
usage....9
Static initializers....20
Streams....22

T

Tags....4
definition....44
proposed....35
Thirty−second rule....11

U

UML....44
Underscores

Copyright 1997 AmbySoft Inc.

use of 2
Users
working closely with 34
V
Visibility
definition..... 44
of accessor methods..... 19
of attributes 14
of methods..... 7
W
Whitespace..... 11
definition..... 44
Why
documentation of..... 3, 9
Widget ..... *See* component

Copyright 1997 AmbySoft Inc.