eSciDoc Development Environment

Coding Standards

Version: 2.1

Author: Roland Werner (ROW), Accenture

Last Changed: 05.09.2007

Revision History

Version	Date	Author	Comments
0.1	14.12.2005	ROW	Initial document
1.0	16.12.2005	ROW	Applied comments from Rolf Vosskamp. Updated version for release.
1.1	19.12.2005	ROW	Applied comments from Lutz Horn.
1.2	21.12.2005	ROW	Applied comments after team meeting.
1.3	13.02.2006	ROW	Reviewed for sending to ZIM.
1.4	18.07.2006	BRP	Adapted for using by ZIM.
1.5	19.07.2006	DOM	Revision of adapted document
1.6	20.09.2006	BRP	Filename and Repository changed. Package section enhanced.
1.7	28.09.2006	DIT	Applied comments after team meeting.
1.8	28.09.2006	DOM	Review of changes after team meeting
1.9	10.10.2006	DIT	Applied comments after review
2.0	17.01.2007	MUJ	Review of Documentation chapter.
2.1	05.09.2007	FRM	Review of changes after team meeting

Referenced and related documents:

Table of Content

1 Objectives

This document outlines standards for programming Java code to improve development quality, readability, and to ease maintenance by making the code more understandable for others.

The developer should make every attempt to adhere to the established criteria. These standards are important for various reasons:

- The majority of the cost for code over its lifetime is in maintenance
- Seldom is a piece of code modified only by its original author
- Standard code is easier and quicker to read and understand for others

The document from FIZ Karlsruhe is adopted for using it in the implementation phase of the eSciDoc applications “Publication Management” and “Scholarly Workbench”.

2 General Principles

2.1 Adhere to the Style of the Original

In general, follow the rules and guidelines documented here. However, if you make changes in a module that obviously was coded according to different rules and guidelines, follow those while changing that code. Following different styles within a single source file makes it more difficult to read and understand.

2.2 Adhere to the Principle of Least Astonishment

To minimize the chances that a user will encounter something surprising in the software, adhere to the principle of least astonishment. To achieve this, follow these main principles while developing:

Simplicity	Build simple classes and simple methods.
Clarity	Ensure each class, interface, method, variable, and object has a clear purpose. Explain where, when, why, and how to use each.
Completeness	Provide the minimum functionality that any reasonable user would expect to find and use. Create complete documentation.
Consistency	Similar entities should look and behave the same. Create and apply standards whenever possible.
Robustness	Provide predictable documented behaviour in response to errors and exceptions. Do not hide errors.

2.3 Follow the Standards in all Code you Produce

Often code that was originally produced only for testing makes its way into the final product.

Apply the rules and guidelines in all code you produce, not only in code for production.

2.4 Document any Deviations

In general, you are obliged to follow the standards described in this document. Before you decide to ignore a rule, make sure you understand why the rule exists and what the consequences are if it is not applied.

If you decide to violate a rule, then document, why you have done so.

3 Naming standards

3.1 Java Naming Standards

There are generally accepted naming conventions employed in Java programming which should be followed. These are general rules to follow when naming classes and their attributes.

3.1.1 General Standards

Each class, attribute, and method name should be meaningful and descriptive of the information it contains and/or the behavior it performs (i.e., addNewCustomer, calculateTotalAmount). Names may not contain spaces **nor may they contain abbreviations**.

All names must be fully spelled out and are strictly in English, except in the case of acronyms. However, acronyms must be treated as normal words e.g. CORBA should be used as Corba. By treating acronyms this way, they fit into all existing naming standards e.g. asCorbaObject, corbaReference etc. If a name is comprised of multiple words, each new word is capitalized (i.e., BidLine, documentCode).

Different from abbreviations, acronyms are allowed.

In general, whitespaces in the code should be used sparsely. No withespace should be used after an opening parenthesis (“(“) / bracket (“[“) or before a closing parenthesis (“)“) / bracket (“]”).

Whitespaces should be used to separate parameters in a method declaration/call, before and after a brace (“{“), before and after an equal sign (“=”) and in computational statements around the operators (“+”, “-“, “/”, “*” etc.).

Examples

- `SecurityFramework`
- `getAccountNumber(accountName)`
- `float rate[i] = (conversion * months) / years;`

3.1.2 Packages

Package names grouping **functional** classes should be preceded with the prefix:

```
de.mpg.escidoc.
```

Package names grouping **test** classes should be preceded with the prefix:

```
test.
```

The prefix is followed by the application-, component- or architecture service name.

All package names must be entirely in lowercase.

3.1.3 File Names

Classes must be named the same as the files in which they are defined. The filename for each file should be the public class name with the correct capitalization followed by ".java". (e.g. ContextManager.java should contain the public class ContextManager).

3.1.4 Class Names

Separate multi-word type names using capitalization with no intervening underscores.

Example

- `ProcessParticipant`
- `SessionManager`
- `HyperLinkLabel`

3.1.5 Method Names

Unlike class names, begin methods with a lowercase and separate multi-word method names using capitalization (e.g. *calculateElectricalLoad*()), except for the constructor which must have the same name as the class name with the same capitalization (e.g. *Settlement*()). Use simple, clear names for methods to allow similar operations across different classes to be named consistently.

Instance methods should simply name the operation they perform on their associated object. Methods used to read or write properties (getters and setters), should start with "get" or "set" respectively.

Example

```
private String getParticipantName();
private void setParticipantName(String newParticipantName);
```

To indicate a boolean value returned by a method, name the method *is*XXXX. This rule does not apply for classes conforming the JavaBeans Specification where *get*XXX is required even for boolean values.

To indicate the setting of a boolean value, name the method *set*XXX(arg).

Example

```
private boolean isValid();
private void setValid(boolean valid);
```

Since instance methods are always invoked in the context of an object (with the exception of static methods), their names should not encode the class name or the type of arguments expected. As an example, a String class instance function that returns the length of the string should be simply named *length*() — it should not be named *stringLength*(), since the context makes the meaning clear.

Avoid the use of Object/object in any method name.

Static methods follow the same conventions as other methods.

3.1.6 Member Variables Names

Variable names always begin with a lower case letter (i.e., documentCode, name, identifier). Variables may be polymorphic across classes, yet must be unique to its own class (For example, Bid, BidLine, and Customer classes may each have the attribute 'identifier', which is unique to that class).

- Always declare variables private. When exposing an instance variable to another object, declare the variable private and provide public or protected methods for accessing the variable.
- Use descriptive names for variables starting with a lower case and starting new words with upper case.
- A variable name should **not contain its own class name** (Bid should not contain a variable 'bidIdentifier'. The variable should be named 'identifier'. This will prevent subclasses from inheriting variables that refer to the parent class).
- **No prefix** is used for local variable names.

Example

```
public class MyClass
{
    private String inputBuffer;

    public final void setInputBuffer(String inputBuffer)
    {
        this.inputBuffer = inputBuffer;
    }

    public final String getInputBuffer()
    {
        return this.inputBuffer;
    }

    ...

}
```

3.1.7 Local Variable Names

Separate multi-word local variable names using capitalization with **no** intervening underscore. Local variables should start in lowercase, and variable names should be descriptive of what the variable's purpose is.

- **No prefix** should be used for local variable names.

- **Use business related names,** i.e. use comprehensible names that demonstrate the role fulfilled by the object, e.g. if it is a new customer object use newCustomer. This removes ambiguities when there is more than one parameter of the same type e.g. 'newCustomer', 'existingCustomer' rather than 'aCustomer1', 'aCustomer2' etc.

- **Never create a temporary variable named "temp"**. Instead use a name that indicates what the variable is needed for.

- **Collections should be pluralized** (i.e., ending in "s") as in customerAccounts or activeThreads.

- **Avoid "hiding" names**. Name hiding refers to the practice of naming a local variable, argument, or attribute the same as that of another one of greater scope. For example, if you have an attribute called *participantName* don't create a local variable or parameter called *participantName*. This makes code difficult to understand and more prone to bugs.

  An exception to this rule should be the definition of setter methods, where the parameter should have the same name as the variable, which is about to be set (see example). The reason for this is that it is normally done in that way in the Java community and that the standard settings of Eclipse's auto-generation create the setter methods this way.

Example

```
public void createMeterReading()
{
    int  counterIndex;
    Account[] customerAccounts;
    Account customerAccount;

    <statements>
}

public void setName(String name)
{
    this.name = name;
}
```

3.1.8 Constant Names

Use upper case for all constants, using underscores to separate multiple words. Constants are Properties that are defined final static and are of either a simple type or String.

Example

```
static final int SPEED_OF_WATER = 3.10;
public static final Logger logger = Logger.getLogger("myclass");
instance = new Singleton();
```

3.1.9 Arrays

There are two styles of declaring an array in Java:

```
int integerArray[] = { 1, 2, 3, 4, 5 };
```

```
int[] integerArray = { 2, 4, 6, 8 };
```

Use the latter style; i.e., the int[ ] integerArray style.

3.1.10 Exception Names

Exception names should end with the word “Exception”. Separate multi-word exception names using capitalization with no intervening underscore and start with initial capitalization.

4 General Java Coding Standards

4.1 Java Program Organization

Coding standards and guidelines should be followed to ensure consistency. The implementation of standardized program organization is critical to achieving quality and consistency objectives.

4.1.1 Class Structure

Only one top-level class can be defined per file.

4.1.2 Class Organization

Organize code units as follows: (a unit is a file)

- beginning comments
- *package* package_name
- *import* class_names
- class/interface documentation comment
- *class/interface* statement
- inner class definitions
- class/interface implementation comment, if necessary
- constants (using static final)
- class variables
- instance variables
- constructors
- finalize()
- class methods
- instance methods
    - getters / setters
    - instance methods

Notes:

This format should be followed unless program functionality, logical relations or viewability dictates otherwise.

1. Instance variables and static variables except of constants should only be private. Have public getters and setters for those that you wish to make public.

2. Define instance methods in a logical order (as determined by the developer) for each static and instance section.

3. Only import the classes that are necessary for the program. (e.g. If your program references the class Component, you should only import java.awt.Component rather than import java.awt.*)

4. Getter and setter functions of the same scope (e.g. public) should be defined adjacent to each other in pairs.

4.1.3 Class Header

The class header consists of the package name and the import section. In the import section, list each imported class explicitly.

Example:

Right	Wrong

`import java.awt.Frame;`
`import java.awt.Graphics;`
`import java.awt.event.WindowAdapter;`
`import java.awt.event.WindowEvent;`
`import java.applet.AppletContext;`
	`import java.awt.*;`
	`import java.awt.event.*;`
	`import java.applet.*;`

In addition, there should be a comment block describing the class and the author. This comment block should be in a format suitable for generating JavaDoc documentation. The comment section also contains the Subversion-specific keywords Author, LastChangedDate and Revision, which will automatically be filled in by Subversion when the file is commited.

```
/**
 * This is a description of the class.
 * The first line of the description should
 * show a summary of the class, and subsequent
 * lines a detailed description.
 *
 * @author:  $Author$, created 14.12.2005
 * @version: $Revision$ $LastChangedDate$
 */
```

Additionally a (non-JavaDoc) comment section can be included after the class/interface declaration, which documents important steps (i.e. major changes/addition; simple bugfixes etc. can be omitted) or implementation details.

4.1.4 Code Layout

A good layout strategy should accurately and consistently represent the logical structure of the code, it should make the code readable, and it should be easy to maintain. The rules in this section are designed to meet those criteria.

4.1.5 Class Headers

- Write class headers on a single line if there is room for it.
- If not, break the line before extends and implements. Indent succeeding lines.
- Put the opening brace in the next line of the class header.

Example

```
public class OvernightAirDeliveryItem
    extends DeliveryItem implements Serializable
{
```

4.1.6 Method Headers

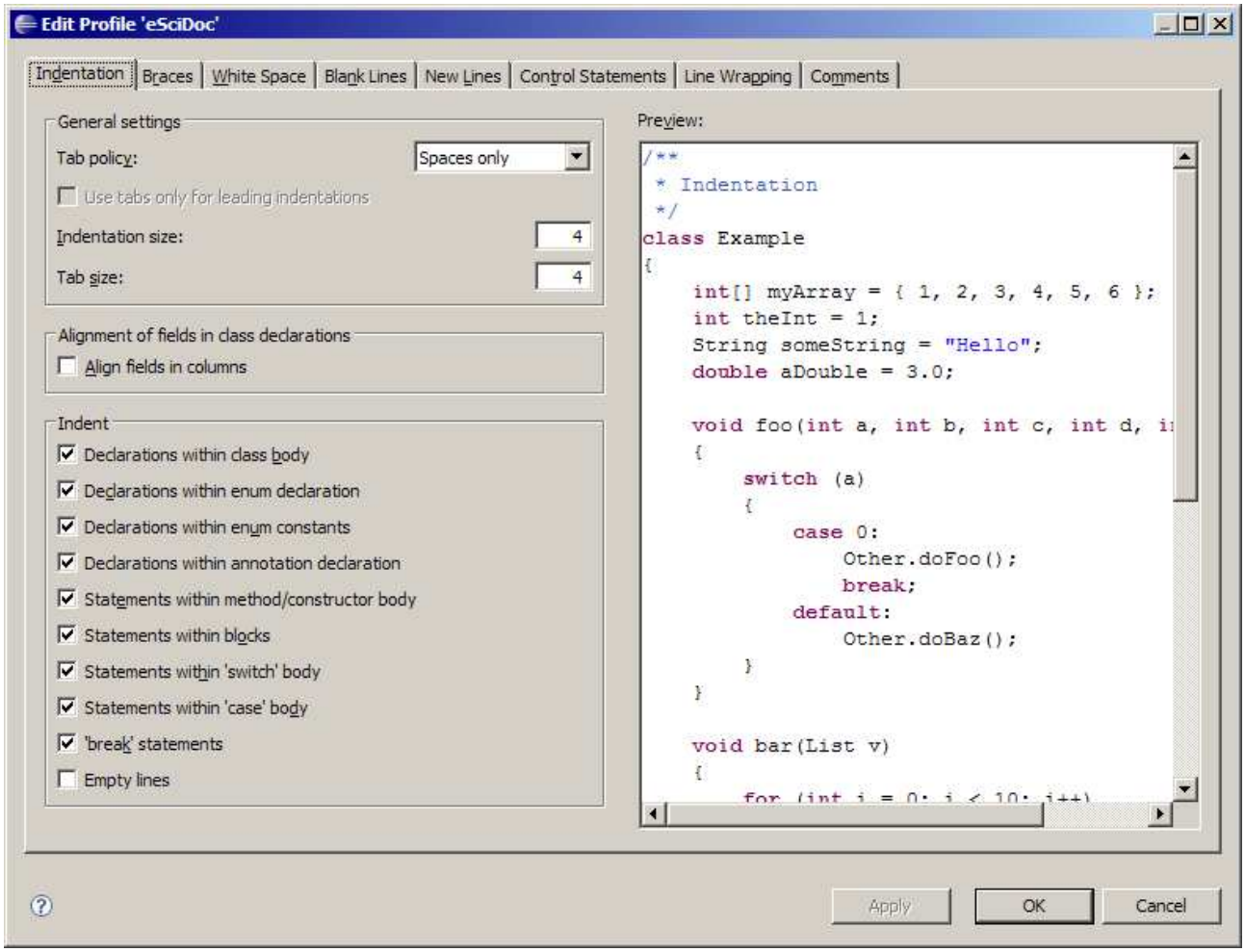
- Align parameters if distributed over several lines.
- Make each parameter final if there is no good reason against it.
- Put the opening brace in the next line of the method header.

Example

```
public boolean isReadyToBeShipped(final boolean paidFor,
                                  final boolean orderFilled,
                                  final boolean addressVerified)
{
```

4.1.7 Indentation

Indentation is 4 blanks. You can configure the IDE to automatically replace tabs by blanks. In Eclipse this can be done in 'Window' | 'Preferences…' by navigating to 'Java' → 'Code Style' → 'Formatter', and creating a new profile (in case a built-in profile is used) or changing a existing one. The 'Tab policy' should be set to 'Spaces only' and the Indentation size as well as the Tab size should be set to '4' (see screenshot below).



4.1.8 Braces

Always use (curly) braces, even for blocks with only one statement. This removes one common source of bugs and eases maintenance:

1. You can insert or remove statements within a block without worrying about adding or removing braces.

2. You never have a problem matching else clauses to if clauses.

Example:

Right	Wrong

`if (clickedRow < currentIndex)`
`{`
`  myTopRow = currentIndex + 1;`
`}`
`else if (currentIndex < myTopRow)`
`{`
`  myTopRow = currentIndex;`
	`if (clickedRow < currentIndex)`
	`  myTopRow = currentIndex + 1;`
	`else if (currentIndex < myTopRow)`
	`  myTopRow = currentIndex;`

}	

This rule applies to the following constructs:

- for, while and do-while loops
- if-else statements
- try, catch and finally clauses
- synchronized blocks

4.1.9 Line Lengths and Line Breaks

- One statement per line.
- If you must break a line, indent the continuation line(s).
- Break lines at least after a maximum of 120 characters.
- If you must break a line, make it obvious by ending the first line with something that needs a continuation:
  1. Break assignments after the assignment operator.
  2. Break arithmetic and logical expressions after an operator.
  3. Break the line to emphasize major sub-expressions.
  4. Break method invocations after the opening parenthesis. If the parameter list still won't fit, break between each parameter or between each logical group of parameters if this seems better.
  5. If you need to break conditional expressions (e.g., in if or while-statements), follow rule 2 above.
- Using extra variables to hold partial (intermediate) expressions can help you avoid line breaks and at the same time improve readability by making the code self-documenting.

Original condition

```
if (clickTime - myPreviousClick < DOUBLECLICK_TIME &&
    mySelection == rowClicked)
{
    ...
}
```

Possible rewrite

```
final long clickInterval = clickTime – myPreviousClk;
final boolean doubleClick = (clickInterval < DOUBLECLICK_TIME);
final boolean clickedSameRow = (mySelection == rowClicked);

if (doubleClick && clickedSameRow)
{
    ...
}
```

4.1.10 Switch/case Layout

- Indent each case one tab level from the switch.

4.2 Java Programming Style

This section integrates the standards and guidelines defined above into an overall programming style. This style should be emulated as closely as possible while developing Java applications.

4.2.1 Class Declarations

Summarized from above, the following is an example of a class declaration (without documentation):

```
public class Circle
{
    public static final float PI = 3.14F;
    private float area;
    private float radius;

    public Circle()
    {
        this(1);
    }

    public Circle(float radius)
    {
        super();
        this.radius = radius;
        calculateArea();
    }

    public float getArea()
    {
        return area;
    }

    public void setArea(float area)
    {
        this.area = area;
    }
```

```
        public float getRadius()
        {
            return radius;
        }

        public void setRadius(float radius)
        {
            this.radius = radius;
            calculateArea();
        }

        private void calculateArea()
        {
            setArea((float)Math.pow(PI * radius, 2));
        }
    }
```

4.2.2 Abstract Methods

Use abstract methods for classes which will not be instantiated i.e. which are to be superclasses for some class hierarchy. Only define abstract methods for operations that need to be defined in derived classes. Interface methods (for example) are implicitly abstract. If a class implements at least one abstract method, the class should be explicitly declared as abstract.

4.2.3 Visibility

Be as restrictive as possible when setting the visibility of a method. If a method doesn't have to be public, then make it protected, if it doesn't have to be protected, make it private. Use the table below as a guide.

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 accessed 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 it was defined, any subclasses of that class, or any class in the same package as the defining class.	When the method provides behavior that is needed internally within the class hierarchy but not externally.
package (or default)	A method defined with package (or default) protection is accessible to all the other classes in the same package, but not outside that package.	Be very careful when using this visibility type.
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 know as reorganizing, the behavior of other methods within the class to encapsulate one specific behavior.

4.2.4 Method Size and One Screen Rule

In order to maintain maximum readability and maintainability, methods should strive to conform to the “One Screen Rule”, meaning that the method is one screen or fewer statements long. This is achievable using function decomposition and using helper functions. Methods should not be longer than 100 lines.

A method should preferably do one thing, and the method name should reflect this accurately. If it does more, ensure that this is reflected in the method name. If this leads to an ugly method name, reconsider the structure of your code. If you had a function named `initPanelManagerAndReadAccountList`, the code would probably benefit from a split into methods named `initializePanelManager` and `readAccountList`.

4.2.5 Exceptions

Generate exceptions for any error conditions not handled by Java or any other exception handlers. A rule of thumb is to generate errors early and often. It is better to be too extensive in error handling rather than assuming that another portion of the code will catch errors.

Handle exceptions at appropriate hierarchical level. Each level of an application hierarchy should cope with as many errors as it can. However, each level should pass errors it can't cope with to

higher levels. A level should not call methods from any higher levels when dealing with an error. This introduces cyclic and unnecessary dependencies between levels of an application. Where applicable, **group exceptions** via an exception hierarchy. All exceptions should be based on a hierarchy.

A sub-system of an application or a service that has (or may have in the future) several different types of errors should map each error with distinct names derived from a generic base. This allows a user to catch all exceptions at a specific level without having to list all the exception types.

Example:

A function may call a File Access framework that throws the following execptions: FileNotFoundException, FileLockedException and EndOfFileException. The calling function may decide to handle all the exceptions from the IOException or handle each one individually.

4.2.6 Bracing and Nesting Styles

For all classes, the declaration shall begin in the leftmost column. The left brace should be positioned in the next line as the class declaration and the right brace for the class must also appear in the leftmost column.

In general, matching beginning braces should be positioned in the next line after the control statement and ending braces should also be positioned in the same column as the statement. This makes identifying statement blocks easier. All declarations and code inside the block must be indented at least one level. A level is defined as 4 spaces. Ensure that your development environment is set up to convert tabs to spaces.

Programming constructs must have the following form:

For loops:

```
for (<initialization>; <condition>; <increment>)
{
    <statements>
}
```

While loops:

```
while (<condition>)
{
    <statements>
}
```

If constructs:

```
if (<condition>)
{
    < statements >
}
```

If … else constructs:

```
if (<condition>)
{
    < statements >
}
else
{
    < statements >
}
```

If … else if constructs:

```
if (<condition>)
{
    < statements >
}
else if (<condition>)
{
    < statements >
}
else
{
    < statements >
}
```

Switch constructs:

```
switch (<variable>)
{
    case <value>:
        <statement>
        <statement>
        break;
    case <value>:
        <statement>
        <statement>
        break;
    default:
        <statement>
}
```

Try … catch constants:

```
try
{
    < statements >
}
catch (Exception_Type variableName)
{
    < statements >
}
```

4.2.7 Variables (general usage)

Declare only one variable per line of code.

Right	Wrong
`private int myWidth = 150;` `private int myHeight = 50;`	`private int myWidth = 150,` `myHeight = 50;`

Also, declare variables in semantic, not alphabetical, order.

Example

`houseNumber`
`streetName`
`city`
`zip`
`country`
`inPostalCode`
`outPostalCode`

4.2.8 Instance Variables

All variables should be initialized before they are accessed. Initialization may occur in either a constructor or as a result of *lazy initialization*. Lazy initialization is the act of initializing variables in their getter methods. This ensures that a variable is not initialized until it is actually needed. This is especially helpful when the object to be retrieved is kept in persistent storage and could be very expensive to build.

4.2.9 Local Variables

Use local variables to represent one thing only. In other words, do not reuse local variables within a method. Whenever a local variable is used for more that one representation you make your code more difficult to understand. The chance of bugs introduced to your code by other developers also increases. Instead, declare a new descriptive variable for use.

Right	Wrong

`int accountIndex;`
`...`
`for (accountIndex = 0;`
`  i < myAccountList.size();`
`  accountIndex++)`
`{`
`  ...`
`}`
`...`
`// Swap elements`
`int arrayElement;`
`arrayElement = intArray[0];`
`intArray [0] = intArray [1];`
`intArray [1] = arrayElement;`
	`int i;`
	`...`
	`for (i = 0;`
	`  i < myAccountList.size();`
	`  ++i)`
	`{`
	`  ...`
	`}`
	`...`
	`// Swap elements:`
	`i = intArray [0];`
	`intArray [0] = intArray [1];`
	`intArray [1] = i;`

The two uses of **`i`** above on the right have nothing to do with one another. Creating unique variables for each purpose makes your code more readable.

4.2.10 Collections

Each attribute collection should implement the following getters and setters:

Method Type	Naming Convention	Example
Getter for the collection	get<CollectionName>()	getMeterReadings()
Setter for the collection	set<CollectionName>()	setMeterReadings()
add an object into the collection	add<CollectionItem>()	addMeterReading()
remove an object from the collection	remove<CollectionItem>()	removeMeterReading()

4.2.11 Initialization

1. All variables, including instance and class variables, should be initialized at the point of declaration if possible.

2. Java allows initialization of arrays using the same syntax as C and C++, by enclosing a comma-delimited set of values in braces.

3. Java >=1.4 allows *initializer blocks* among the declarations. An initializer block is a section of code enclosed in braces. There are two kinds of initializer blocks: *static* and *instance*.

Static initializer blocks are executed the first time a class is loaded by a ClassLoader. **In general static initialization blocks should be avoided for any nontrivial initialization task.** The reason for such judicious use is static initialization block errors are *extremely* difficult to debug. Error handling is difficult within a static initializer, and as a result static initializer blocks should be avoided if possible. During static initialization (class initialization), things happen in the following order:

1. Class initialization of the superclass is performed, unless it has been done earlier.

2. Static variables are initialized and static initializer blocks are executed. This happens in the order they are listed, from top to bottom. Instance variables, instance initializer blocks and methods don't figure into this.

Note that static and instance initializer blocks are allowed in Java >=1.4. Static initializer blocks are executed in order when the class is first instantiated; instance initializer blocks are executed in order *after* the superclass constructor runs, but *before* the class constructor runs.

Instance initializer blocks are executed whenever a class is instantiated. During object initialization (instance initialization), things happen in the following order:

1. If this is the first time the class is instantiated, all the class (static) initialization takes place.

2. We enter a constructor. If we have not specified a constructor, the compiler supplies a default constructor with no arguments automatically.

3. The superclass constructor is called. If your constructor does not explicitly invoke a superclass constructor, the default (argument-less) superclass constructor is called anyway.

4. All instance variables are initialized and instance initializer blocks are executed. This happens in the order they are listed, from top to bottom. Class variables, class initializer blocks and methods don't figure into this.

Use initializer blocks to perform any initialization that can't be performed by direct variable initialization; put each initializer block immediately following the variable in question. In the examples below, note that the array can be initialized without using an initializer block, while the vector object requires one because of the calls to the `addElement` method.

Example

```
private Vector listOfSomething = new Vector();
{ // Instance initializer block
    listOfSomething.addElement(someObject);
    listOfSomething.addElement(anotherObject);
}

private static int[] multipliers = {
    5, 4, 3, 2, 7, 6, 5, 4, 3, 2};

private static MyClass myClass = new MyClass();
static
{ // Static initializer block
    myClass.setValue(someValue);
}
```

4.3 Documentation for Java Programs

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. Well documented code is readable and easier to maintain. When commenting code, you should:

- Only comment confusing/interesting calls
- Ensure that comments are valuable, not redundant

4.3.1 Comments

Java has three styles of comments: Documentation comments start with /** and end with */, C-style comments which start with /* and end with */, and single-line comments that start with // and go until the end of the source-code line. In the chart below is a summary of the suggested use for each type of comment, as well as several examples.

Usage	Example

Use documentation comments directly before declarations of interfaces, classes and methods to document them. Documentation comments are processed by javadoc, see below, to create external documentation for a class.	`/**`
	` * A participant is`
	` * any organization who bids on`
	` * energy or submit meter data.`
	` *`
	` *  @author  $Author$`
	` */`

Use single line comments internally within methods for single line	`// Do a double-flip.`

comments or to comment out sections of code.	`...`

Use block comments to describe implementation details like algorithms or data structures	`/*`
	` * Here is a block comment with`
	` * some very special formatting.`
	` *`
	` * one`
	` *  two`
	` *  three`
	` */`

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. You should refer to the Javadoc standards in the following section and the JDK javadoc documentation for further details.

4.3.2 Classes

Every class or interface definition should be preceded by a javadoc description. This description should follow the format listed below:

Example

```
/**
 * This class blah...
 *
 * @author Foo Bar (initial creation)
 * @author $Author$ (last modification)
 * @version $Revision$ $LastChangedDate$
 */
```

Each class has to have two @author-tags in the header. One to identify the initial creator of the class, and another to identify the person who did the last modification of the class. All following methods are written by the initial author, unless a method has an @author-tag of its own.

4.3.3 Methods

Each method that has a scope wider than "private" must have a specification block associated with it.

Method specification format:

```
/**
 * <Method description>
 *
 * @param   parameter-name description
 * @return  return-description
 * @exception  exception-description
 */
```

Major changes of existing methods should be documented in the code by adding the name of the editor followed by a detailed description of what has been changed in this method. If the changes belong to a documented bug the bug ID should be added as well.

Example

```
/**
 * This method will assign a new Location
 * to the role.
 *
 * @param location the new Location
 */
public void setLocation(Location location)
{
    ...
}
```

4.3.4 Variables

Although variable names should be chosen to indicate their purpose, if needed for clarity an inline comment (//) describing the variable can be placed on the line with the variable declaration.

Example

```
Public class Window
{
    ...
    private Window parent; // the Window I was created from
```

4.3.5 White Space

Use a blank line to separate logical groups of code and method definitions.

Example

```
public void addCustomer(Customer newCustomer)
    throws InvalidCustomerException
{
    if(newCustomer.isValidated())
    {
        getCustomerList().add(newCustomer);
    }
    else
    {
        throw new InvalidCustomerException();
    }
}

public Customer getCustomer(int customerKey)
{
    return getCustomerList().getElement(customerKey);
}
```