

Java Coding Standards

Bill Rushmore
rushmore230@charter.net

Who am I and why am I here?

- Java Developer since 1997
- Started a Java User Group in 1998
- Just a “regular” Java guy obsessed with creating top quality Java code.
- Started a Special Interest Group to discuss Java Coding Standards that lead to the open source book *Coding Standards for Java*

New England Java Users Group (NEJUG)

- Founded in November of 1998
- Typically Meets at Sun's East Coast Headquarters
- Over 2600 members
- Another one of the top 25 Java User Groups!

NEJUG Coding Standards Special Interest Group

- Since group become so large discussion between developers become difficult so “SIGs” were started
- Coding Standards SIG first met in January 2001
- Quickly the group decided to share ideas with the rest of the user group and Java community

“Coding Standards for Java”

- After a year worth of work the book “Coding Standards for Java” was released
- Released as open source and can be down loaded off of the NEJUG site: <http://www.nejug.org>
- Redistribution is encouraged
- Recently translated into Spanish
- Used by several university classes

Why Yet Another “Standard”?

- Little or no explanation of why
- Too many contradictions
- We wanted to give developers options
- We wanted to make a tool useful in everyday work

Some of the Others...

- *Elements of Java Style*, Cambridge University Press
- Code Conventions, Sun Microsystems,
<http://java.sun.com/docs/codeconv/>
- Try to Google “Java Coding Standards”...

Why Standards Are So Important?

- Improves readability – Code should read like a good book
- Most of software costs goes to maintenance
- Improves overall quality, “The Broken Window Theory”

Standards, Styles, and Conventions

- Standards:
 - Universal Acceptance
 - Little tolerance for violations
- Styles:
 - Recommendations
 - Expect some disagreements
- Conventions
 - Good Practices

Standards

STD-1: Package Naming

- Packages should be lower case:

This:

```
package java.util;  
package com.myapp.mypackage;
```

Not This:

```
package javax.MyPackage  
package Com.MyPackage
```

STD-2: Class and Interface Naming

- Classes and Interfaces should be nouns and upper case (Pascal Case)

This:

Object

Customer

Not This:

myClass

My_Class

MYCLASS

STD-3: Method Naming and Formatting

- Methods should be verbs and start as lower case (Camel Case)

This:

`getX()`

`createX()`

Not This:

`Log()`

`CREATE_X()`

STD-4: Variable Naming

- Use camel case for naming

This:

customerId

speed

Not This:

Customer_id

Speed

STD-5: Constant Naming

- All upper case with underscores between words

This:

MAIL_SERVER

MAX_SIZE

Not This:

mailServer

Mail_server

STD-6: Use of JavaDoc is Required

- The public interface is the key to being able to use a class properly
- Not optional for public classes and methods

STD-7: Use of Implementation is Required

- Comments should describe the “why”
- How and where of comments are a matter of styles and conventions, but the need for them is not in question

STD-8: Consistency of Formatting is Required Within a Source File

- Stick with the last developers style, even if you don't like it!

STD-9: Avoid Local Declarations Which Obscure Declarations at Higher Levels

- Do not use “this”:

```
private int stuff;  
  
public void setStuff(int stuff)  
{  
    this.stuff = stuff;  
}
```

Styles

STY-1: Order Sections Within a Source File Consistently

1. Package or file level comments
2. Package and import statements
3. Public class or interface declaration
4. Private class and interface declarations

STY-2: Ordering of Import Statements

1. Standard packages (java.io, java.util, etc.)
2. Third party packages such as com.ibm.xml.parser
3. Your own packages

Within each group order the packages in alphabetic order

STY-3: Import Statement Detail

Use the wildcard (*) to reduce the import

```
java.util.*;
```

Or...

Do individual imports

```
java.util.Date;
```

```
java.util.Vector;
```

STY-4: Ordering of Class Parts

1. Javadoc comments
2. Class declaration statement
3. Class-wide comments
4. Class static variable declarations (public, protected, package, private)
5. Class instance variable declarations (public, protected, package, private)
6. Methods declarations

STY-5: Ordering of Methods within Classes

- Constructors first
- Functional or alphabetical ordering

STY-6: Limit Length of Code Lines

- Limit lines to 80 characters

STY-7: Line Continuation of Method Signatures

- Double the indentation of the next line

```
public void doSomething(String arg1,  
    String arg2, String arg3)  
{  
    //Stuff goes here  
}
```

STY-8: Line Continuation of General Code

- From Sun's Coding Conventions:
 - Break after a comma.
 - Break before an operator.
 - Prefer higher-level breaks to lower-level breaks.
 - Align the new line with the beginning of the expression at the same level on the previous line.
 - If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 8 spaces instead.

STY-9: Indentation Levels

- Use a consistent number of spaces for each indent
- 2,3,4, or 8
- Just pick one and stick to it!

STY-10: Indentation Using Tabs

- Don't use hard tabs

STY-11: Indentation of Controlled Statements

- Indent the statement block of a compound statement

STY-12: Brace Placement

The great debate! Where to put the '{'

End of line:

```
if (stuff == 0) {
```

The next line:

```
if (stuff == 0)  
{
```


STY-13: Ternary Statement Usage

- Limit the use of the `?:`, they can be difficult to read
- Limit their use to single line simple cases
- Never nest them!

STY-14: Always Use a Break Statement in Each Case

- Avoid the unexpected fall through!

```
switch(test)
{
    case 0:
        //do something here
        break;
    case 1:
        //do something here
        ...
}
```

STY-15: Include a Default Case in All Switch Statements

- Code defensively!

```
switch(test)
{
    case 0:
        //do something here
        break;
    ...
    default:
        System.out.println("Unexpected value!");
}
```

STY-16: Initialize Variables where they are declared, But Only for Non-Default Values

This:

```
Class Foo
{
    private Thread myThread = new Thread();
    private speed;
```

Not This:

```
Class Foo
{
    private Thread myThread;
    private speed=0
```

STY-17: Initialize Members and Sub-Objects either in a Declaration or in Constructors

- Pick either the constructor or the declaration to declare variables, not both
- Easier to follow and eliminates duplication

STY-18: When Commenting Out Code, Only Use // Style Comments

- No need to worry about nested comments
- Can be misleading if a large group of code is commented, (if your ide doesn't color-code)
- Use your change management instead of commenting out large sections

STY-19: Properly Format Comments

1. A comment block should be preceded by a single blank line
2. A comment should precede the code to which it relates
3. A comment should be indented to the same level as the code it relates to.

STY-20: Comments Should Not Obscure the Code

- I.E. Don't have too many comments
- Use a comment for a whole code block rather for each line

STY-21: Variable Declaration Grouping

1. One declaration per line
2. Order the declarations in some fashion.
3. New declarations should go into their appropriate place according to the ordering being used. A comment should indicate when and why the new variable was added.

STY-22: Place Variable Declarations at the Beginning of the Innermost Enclosing Block

- You can declare variable anywhere but keep them in a predictable place
- Class variables at the top of the class
- Method variables at the top of the method
- Block variables at the top of the block
- For loop variable can be declared in the statement

STY-23: Limit the Number of Java Statements per Line to 1

- Multiple statement can hide code to the casual observer
- Makes stepping through code difficult

STY-24: Optional Braces are not Optional

- Makes code easier to read
- You never know if you want to add another line of code!
- Avoids bugs from unintentional results

STY-25: Parameter Naming

- Make sure your parameters mean something

This:

```
public double calculate(double totalPrice, double units)
```

Not This:

```
public double calculate(double a, double b)
```

STY-25: Parameter Naming (cont)

All your variables should have meaningful names!

String number = "";

String numbe = "";

String numb = "";

String num = "";

String nu = "";

STY-26: Method Naming for Assessor Methods

- Follow the Java Bean Standard

```
public int getValue()
```

```
public void setValue(int valueIn)
```

STY-27: Use Prefixes to Indicate Variable Scopes and Sources

- The under score style:
 - Use `_` or `m_` for class variables
 - `l_` for local variables
 - `P_` for parameters
- The single character:
 - Use `g` for global
 - Use `p` for parameter
- No prefixes at all style

STY-28: Use Blank Lines to Organize Code

- Single blank lines
 1. Between local variable declarations and the first code in a method
 2. Before a block comment
 3. Between logical sections of code to improve readability
- Double blank lines
 1. Between methods
 2. Between class and interface definitions
 3. Between any other sections of a source file

Conventions

CON-4: Limit the Length of Methods

- A method should be about a “page of code”
- Around 30 lines of code
- We have all seen, methods hundreds of lines long

CON-5: Limit the Length of Source Files

- Try to stick to something around 2000 lines of code
- Some of the craziest I have seen:
 - 30,000 line statefull session EJB
 - 5,000 line JSP (Not including the headers!)

CON-8: All Class Variables Should Be Private

- Use the gets and sets!
- Enables defensive programming
- Encapsulation is always a good thing

CON-9: Limit the Number of Parameters

- Good rule of thumb is 5 parameters
- If you need more consider using value object parameter
- Worst I have seen is 16, I usually ended up putting the wrong value in the wrong parameter!

CON-12: Avoid Nesting Conditions More Than 3 Deep

- If you get to the point of nested three deep, time to refactor

CON-13: Define Constants in Interfaces

- Allows for reusability
- Easy to find and maintain constants
- Use “wrapper” class when implementations aren't appropriate

Using Standards in Your Projects

- Make sure the entire team agrees on standard
- Automate as much as possible
- Use the NEJUG Coding Standards for Java as your foundation!

Extending Coding Standards for Java

For Example:

1. Use the NEJUG Coding Standards
2. Follow all Standards
3. Adhere to STY-1, STY-3 (using '*' format) etc...

Code Reviews

- Consider not allowing code that does not adhere to standards to pass a code review
- Rarely if ever should styles or conventions be issues during a code review

Automatic Code Formatting

No need to spend ALL your time formatting code!

IDE's

- Make sure your IDE settings meets with your project's rules
- Some projects will standardize on one IDE all with standard settings

Ant

- You automate your testing, automate your formatting!
- See “Ant in Anger”
- Use tools such as Jalopy, Jindent, etc.

The End

Now Go Out There and Write Some Good Code!