Coding Style for Java Programs

Harry H. Porter III
January 24, 2003

What coding "style" am I looking for in your Java programs?

Here are some rules regarding my recommended coding style. For more thoughts and opinions on style, see my paper Harry's Style Guide: Helpful Hints for Programming in the C Language. This is available at:

```
cs.pdx.edu/~harry/musings/Progstyle.html
cs.pdx.edu/~harry/musings/Progstyle.pdf
```

Good style increases program reliability.

The goal of good coding is to produce software that works correctly and that can be repaired or modified easily. With larger programs, the fundamental key is to create programs that are easy to read, both by you and others. Following a set of stylistic conventions, like the rules described here, when you code, goes a long way toward achieving the goal of creating good, readable software.

Every method should have a comment in front of it. The first line should give the name, parameters, and return type, which makes it easy to scan the code. The comment should tell what the method does. It should be written primarily for the benefit of the caller.

Format the method comments like this: The extra lines are useful in setting the comment off from surrounding code. Indent the comment the same amount as the method header.

```
//
// lookupToken (str) --> tokenType
//
// This method looks string "str" up and returns the
// corresponding token type.  It returns -1 if not found.
//
static int lookupToken (String s) {
    ....
}
```

For fields, set them off with a single, one-word comment, like this. Also include a comment for each field describing the field.

```
//
// Fields
//
private String str;          // The key
private int token;           // The value
```

For constructors, use a comment that identifies the method as a constructor only. We know what a constructor does so there is no need to say anything more. Comment each constructor separately.

```
//
//  Constructor
//
StringTable (String s, int t) {
    ...
}


//
//  Constructor
//
StringTable (String s) {
    ...
}
```

For static fields, include a comment like this to make locating them easy. Comment each individual field appropriately.

```
//
// Static Fields
//
static InputStream in;       // Where the input comes from
static int errorCount = 0;   // How many errors, so far
static Lexer lexer;          // The lexer
```

Comment the entire class with a single comment before the class, as shown next. The line of dashes with the class name serves to identify the whole class. This is helpful when scanning a file with multiple classes, or when looking at a printout of the file.

Include your name and the date at the beginning of every file. The beginning of a file might look like this:

```
// --------------------------- Lexer -------------------------
//
// This class is intended to be instantiated exactly once.  The
// instance of this class is a lexical analyzer for a particular
// input source.  The key method is "getToken", which reads and
// returns the next token.
//
// Harry Porter -- 01/15/03
//

import java.io.*;

class Lexer {
    ...
}
```

Within a class, include the parts of a class in this order:

```
static fields
non-static fields
static methods
constructors
methods
```

When indenting code, always **increment in multiples of 4 spaces**.

A series of statements that are executed sequentially must be indented equally.

```
x = 123;
 y = x * a;                  // Very wrong
z = c.meth (x, y);
```

Never use the TAB character. The handling of TABs is not standard; if you stick to spaces, your program will print out nicely on any system. Before you hand in your program, please search for TABs and replace them with spaces, to make sure.

Limit your line lengths to 80 characters. This will prevent line "wrap-around" when printing. Break long lines into multiple lines, rather than letting the lines wrap-around.

Always use braces for IF, WHILE, FOR, etc., even when they are not syntactically necessary.

```
while (...)      // wrong
    x = ...;

while (...) {    // right
    x = ...;
}
```

Do not start a line with an opening brace; put the { on the previous line.

```
if (...)      // wrong
    {...
    }
else
    {...
    }

if (...) {    // right
    ...
} else {
    ...
}
```

As an exception, you can format methods header like this when they throw exceptions.

```
//
// getToken () --> int
//
// This method does...
//
int getToken ()
    throws CompileTimeError, LexicalError, AnotherBooBoo
{
    ...
}
```

Each closing brace should be on a line by itself, with the exception of the brace in front of the ELSE keyword or CATCH keyword.

The closing brace should be in the same column as the first character of the line containing the matching opening brace. For example:

```
while (...) {
    ...
}
```

Put ELSE on a line by itself. Format IF statements like this:

```
if (...) {
    ...
} else {
    ...
}
```

Format IF-ELSE-IF sequences specially. Here is a sequence of nested IF statements, formatted according to the rules:

```
if (c == EOF) {
    return (LX_EOF);
} else {
    if (isAlpha (c)) {
        return (LX_ALPHA);
    } else {
        if (isDigit (c)) {
            return (LX_DIGIT);
        } else {
            printError ("...");
        }
    }
}
```

Sometimes code like this will be clearer when viewed as a sequence of IF, ELSEIF, ELSEIF, ELSEIF, ELSE, END... In this case, format it as shown below. (This is the only time ELSE will not be followed by {.)

```
if (c == EOF) {
    return (LX_EOF);
} else if (isAlpha (c)) {
    return (LX_ALPHA);
} else if (isDigit (c)) {
    return (LX_DIGIT);
} else {
    printError ("...");
}
```

Format TRY statements as shown by this example:

```
try {
    x = ...;
    y = ...;
} catch (CompileTimeError e) {
    ...
} catch (IOException e) {
    ...
}
```

Surround all binary operators by a single space.

```
x = ((a + b) * c) < 100) && (d > 1000);  // right
x=((a+b)*c)<100)&&(d>1000);              // wrong
```

One exception is the PERIOD denoting method invocation or field access.

```
aPerson.birthdate
```

Format message sends as shown next. Spaces around the period are optional and not normally used. Leave a space between the method name and the opening parenthesis.

```
x.foo (...)
```

Never use the assignment operator within expressions. This rule will reduce mistakes due to confusion with ==.

```
if (done = (atEOF || problems)) {        // wrong
    ...
}

done = atEOF || problems;                // right
if (done) {
    ...
}
```

Insert a single space after every COMMA.

```
x.foo (a, b, (c + d), e.foo (x, y))
```

Avoid inserting too many blank lines in code. However, insert 2 or 3 blank lines in front of each method. Also put 2 or 3 blank lines before the fields. Put 2 or 3 blank lines before each constructor.

Try to declare method variables at the beginning of the method, rather than in the middle of the method. Leave a blank line between the method's variables and the first statement, as shown next.

```
void foo (...) {
    int t, totalCount;

    if (...) {
        t = ...;                     // Don't use "int t = ...;" here.
        totalCount = prevCount + t;
    }
    ...
}
```

If a variable is used locally in a FOR, it may be clearer to declare it there, rather than at the top of the method. For example:

```
void foo (...) {
    ...
    for (Iterator it = idList.iterator (); it.hasNext (); ) {
        ...
    }
    ...
    for (int i = 1; i<=100; i++) {
        ...
    }
    ...
}
```

Use identifiers that consist of full English words. For example:

```
tokenBuffer
parseReadStatement
```

The first character of each word is capitalized and all other letters are lowercase. Begin the variable name with a lowercase and avoid the underscore character.

Variable names should be given names that are nouns, like "tokenBuffer" and "whiteSpaceCharacterSet." Methods should be given names that are actions or commands, like "getToken" and "printError." Methods that return true or false should be given names that ask a yes-no question, such that a yes answer corresponds to a true result. For example, the method "isLetter" might used to test a character and return true if it is a letter character. Boolean-valued variables can be given names that form a statement of fact, like "gotLetter" and "atEnd." Variables that are only used over the span of three or four statements can be given single letter names to indicate their temporary nature.

Here is an example where my style conventions are not followed. What does this code do?

```
if (ep)
        erdisp("...");
else {
        t=l.nxt();
        c=tbuf[1];
        if (let(c)) l=true;}
```

Here is the same exact code, following my style conventions. Is this code easier to understand?

```
if (atEnd) {
    printError ("...");
} else {
    nextToken = lexer.getToken ();
    ch = tokenBuffer [1];
    if (isLetter (ch)) {
        gotLetter = true;
    }
}
```

Capitalize all constant names.

```
final static int
    VAR = 0,        // 'var' keyword
    SET = 1,        // 'set' keyword
    ...
```

Avoid Javadoc comments. For CS-321 / 322, we will not be using the "javadoc" tool, so do not include javadoc comments, such as:

```
/** Lexer for the PCAT language.
 *  @author Harry Porter
 */
```

Test your code thoroughly. Execute every line of your code. Test all boundary conditions. If you change your code, always test every change, NO MATTER HOW TRIVIAL. You must execute EVERY line of code and you must verify through testing that every single piece of your program is working.

Understand your code. It is not sufficient to produce code that passes test cases; you must understand how your code works. If your code fails, begin by understanding what it does. Never try to fix a bug without understanding the bug first.

If you smell a problem, investigate immediately. Don't ignore any suspicious behavior. Don't move on to other areas without first being 100% sure that everything is okay.

Adopting a coding style will make coding easier. When coding, there are many decisions to be made. Some decisions are large and complex, but many decisions are trivial and mundane, like when to insert an extra space character. Each of the countless unimportant decisions of formatting your code distracts you a tiny bit from the main task. They will detract from the quality of your code and impinge on the logical correctness of your programming. By adopting these stylistic conventions, you are freeing your mind from the drudge work of coding and allowing it to concentrate fully on the bigger picture, namely the much harder tasks of creating high quality, complex software systems.