12

Coding Standards

CERTIFICATION OBJECTIVES

- Use Sun Java Coding Standards

CERTIFICATION OBJECTIVE

Use Sun Java Coding Standards

The Developer exam is challenging. There are a lot of complex design issues to consider, and a host of advanced Java technologies to understand and implement correctly. The exam assessors work under very strict guidelines. You can create the most brilliant application ever to grace a JVM, but if you don't cross your t's and dot your i's the assessors have no choice but to deduct crucial (and sometimes substantial) points from your project. This chapter will help you cross your t's and dot your i's. Following coding standards is not hard; it just requires diligence. If you are careful it's no-brainer stuff, and it would be a shame to lose points because of a curly brace in the wrong place. The Developer exam stresses things that *must* be done to avoid automatic failure. The exam uses the word *must* frequently. When we use the word *must*, we use it in the spirit of the exam, if you *must* you *must*, so just get on with it. Let's dive into the fascinating world of Java Coding Standards.

Spacing Standards

This section covers the standards for indenting, line-length limits, line breaking, and white space.

Indenting

We said this was going to be fascinating didn't we? Each level of indentation must be four spaces, exactly four spaces, always four spaces. Tabs must be set to eight spaces. If you are in several levels of indentation you can use a combination of tabs and (sets of four) spaces to accomplish the correct indentation. So if you are in a method and you need to indent 12 spaces, you can either press SPACEBAR 12 times, or press TAB once and then press SPACEBAR four times. (Slow down coach.) We recommend not using the TAB key, and sticking to the SPACEBAR—it's just a bit safer.

When to Indent If you indent like this, you'll make your assessor proud:

- Beginning comments, package declarations, import statements, interface declarations, and class declarations should not be indented.

- Static variables, instance variables, constructors, methods, and their respective comments* should be indented one level.
- Within constructors and methods, local variables, statements, and their comments* should be indented another level.
- Statements (and their comments) within block statements should be indented another level for each level of nesting involved. (Don't worry, we'll give you an example.)

The following listing shows proper indenting:

```
public class Indent {

    static int staticVar = 7;

    public Indent() { }

    public static void main(String [] args) {

        int x = 0;

        for(int z=0; z<7; z++) {
            x = x + z;
            if (x < 4) {
                x++;
            }
        }
    }
}
```

Line Lengths and Line Wrapping

The general rule is that a line shouldn't be longer than 80 characters. We recommend 65 characters just to make sure that a wide variety of editors will handle your code gracefully. When a line of code is longer than will fit on a line there are some *line wrapping* guidelines to follow. We can't say for sure that these are a must, but if you follow these guidelines you can be sure that you're on safe ground:

- Break after a comma.
- Break before an operator.

---

* Rules about comments are coming soon!

- Align the new line a tab (or eight spaces) beyond the beginning of the line being broken.
- Try not to break inside an inner parenthesized expression. (Hang on, the example is coming.)

The following snippet demonstrates acceptable line wrapping:

```
/* example of a line wrap */
System.out.println(((x * 42) + (z - 343) + (x % z ))
        + numberOfParsecs);

/* example of a line wrap for a method */
x = doStuffWithLotsOfArgs(coolStaticVar, instanceVar,
        numberOfParsecs, reallyLongShortName, x, z);
```

White Space

Can you believe we have to go to this level of detail? It turns out that if you don't parcel out your blank spaces as the standards say you should, you can lose points. With that happy thought in mind, let's discuss the proper use of blank lines and blank statements.

The Proper Use of Blank Lines Blank lines are used to help readers of your code (which might be you, months after you wrote it) to easily spot the logical blocks within your source file. If you follow these recommendations in your source files, your blank line worries will be over.

Use a blank line,

- Between methods and constructors
- After your last instance variable
- Inside a method between the local variables and the first statement
- Inside a method to separate logical segments of code
- Before single line or block comments

Use two blank lines between the major sections of the source file: the package, the import statement(s), the class, and the interface.

The Proper Use of Blank Spaces Blank spaces are used to make statements more readable, and less squished together. Use a blank space,

- Between binary operators
- After commas in an argument list
- After the expressions in a *for* statement
- Between a keyword and a parenthesis
- After casts

The following code sample demonstrates proper form to use when indenting, skipping lines, wrapping lines, and using spaces. We haven't covered all of the rules associated with the proper use of comments; therefore, this sample does *not* demonstrate standard comments:

```
/*
 * This listing demonstrates only proper spacing standards
 *
 * The Javadoc comments will be discussed in a later chapter
 */


package com.wickedlysmart.utilities;


import java.util.*;


/**
 * CoolClass description
 *
 * @version .97 10 Oct 2002
 * @author  Joe Beets
 */
public class CoolClass {

    /** Javadoc static var comment */
    public static int coolStaticVar;

    /** Javadoc public i-var comment */
```

```
    public long instanceVar;

    /* private i-var comment */
    private short reallyLongShortName;

    /** Javadoc constructor comment */
    public CoolClass() {
      // do stuff
    }

    /** Javadoc comment about method */
    void coolMethod() {
        int x = 0;
        long numberOfParsecs = 0;

        /* comment about for loop */
        for(z = 0; z < 7; z++) {
            x = x + z;

            /* comment about if test */
            if (x < 4) {
                x++;
            }

            /* example of a line wrap */
            System.out.println(((x * 42) + (z - 343) + (x % z ))
                    + numberOfParsecs);

            /* example of a line wrap for a method */
            x = doStuffWithLotsOfArgs(coolStaticVar, instanceVar,
                    numberOfParsecs, reallyLongShortName, x, z);
        }
    }

    /** Javadoc comment about method */
    int doStuffWithLotsOfArgs(int a, long b, long c, short d, int e,
            int f) {
        return e * f;
    }
}
```

How to Care for Your Curly Braces

If you format your curly braces correctly, you can distinguish your exam submittal from all the other Larrys and Moes out there. We know that this is a passionate

topic for lots of folks; we're just letting you know what your assessor will be looking for, so please don't attempt to drag from us what our real feelings are about curly braces.

Curly Braces for Classes, Interfaces, Constructors, and Methods

OK, along with curly braces, we might talk a little about parentheses in this section. The opening brace for classes, interfaces, constructors, and methods should occur at the end of the same line as the declaration. The closing brace starts a new line by itself, and is indented to match the beginning of the corresponding declaration; for example,

```
public interface Curly {

    public int iMethod(int arg);
}


class Moe implements Curly {

    int id;

    public Moe() {
        id = 42;
    }

    public int iMethod(int argument) {
        return (argument * 2);
    }
}
```

Curly Braces for Flow Control (ifs and whiles, etc.)

Your flow control blocks should always be enclosed with curly braces. There are places where the compiler will let you get away with not using curly braces, such as *for* loops and *if* tests with only one statement in the body, but skipping the braces is considered uncivilized (and in fact often leads to bugs when code is enhanced later). For the exam, always use curly braces. Following is an example of how to structure all of the flow control code blocks in Java—pin this baby to your wall!

```
class Flow {

    static int x = 0;
```

```
static int y = 5;

public static void main(String [] args) {
    for (int z = 0; z < 7; z++) {          // for loop
        x = x + 1;
        y = y - 2;
    }

    if (x > 4) {
        System.out.println("x > 4");       // if test
        x++;
    }

    if (x > 5) {                           // if, else
        System.out.println("x > 5");
    } else {
        System.out.println("x < 6");
    }

    if (x > 30) {                          // if, else-if, else
        System.out.println("x > 30");
    } else if (x > 20) {
        System.out.println("x > 20");
    } else {
        System.out.println("x < 21");
    }

    do {                                   // do loop
        x++;
        System.out.println("in a do loop");
    } while (x < 10);

    while (x < 13) {                       // do while loop
        x++;
        System.out.println("in a do while loop");
    }

    switch (x) {                           // switch block
    case 12:
        x++;
        /* falls through */                // see comment at end
    case 13:
        x++;
        System.out.print("x was 13");
        /* falls through */
```

```
            case 14:
                System.out.print("x is 14");
                /* falls through */
            default:
                break;
            }

            try {                                   // try, catch
                doRiskyMethod();
                x++;
            } catch (Exception e) {
                System.out.println("doRisky failed");
            }

            try {                                   // try, catch, finally
                doRiskyMethod();
                x++;
            } catch (Exception e) {
                System.out.println("doRisky failed");

            } finally {
                x = 100;
            }
        }

        static void doRiskyMethod() {
            x = y;
        }
    }
```

You might want those *Exception*s above to be *RuntimeException*s. *javac* does not mind, but *jikes* will give you a warning.

One interesting thing to notice about the example above is the use of the */* falls through */* comment in the *switch* statement. This comment should be used at the end of every *case* block that doesn't contain a `break` statement.

Our Comments About Comments

Earlier we talked about being a team player. The orientation of the exam is to see if you can create software that is readable, understandable, and usable by other programmers. Commenting your code correctly is one of the key ways that you can create developer-friendly software. As you might expect, the assessors will be looking to see if your code comments are appropriate, consistent, and in a standard form.

This chapter will focus on implementation comments; Chapter 17 will cover *javadoc* comments. There are several standard forms that your implementation comments can take. Based on the results of extensive research and worldwide polling we will recommend an approach, which we believe represents the most common of the standard approaches. If you choose not to use our recommendation, the most important thing that you can do is to pick a standard approach and stick with it.

There are several types of comments that commonly occur within source code listings. We will discuss each of them with our recommendations and other possible uses.

Block Comments

Use a block comment in your code when you have to describe aspects of your program that require more than a single line. They can be used most anywhere, as source file or method headers, within methods, or to describe key variables. Typically, they should be preceded by a blank line and they should take the following form:

```
/*
 *this is a block comment
 *it occupies several lines
 */
```

Single Line Comments

Use a single line comment in the same place you would block comments, but for shorter descriptions. They should also be preceded by a blank line for readability, and we recommend the following form:

```
/* this is a single line comment */
```

It is acceptable to use this alternate form:

```
// this is the alternate single line comment form
```

End of Line Comments

When you want to add a comment to the end of a line of code, use the aptly named *end of line* comment. If you have several of these comments in a row, make sure to align them vertically. We recommend the following form:

```
doRiskyStuff();        // this method might throw a FileNotFoundException

doComplexStuff();      // instantiate the rete network
```

It is acceptable to use this alternate form:

```
doRiskyStuff()         /* this method might throw a FileNotFoundException  */

doComplexStuff();      /* instantiate the rete network  */
```

Masking Comments

Often in the course of developing software, you might want to mask a code segment from the compiler without removing it from the file. This technique is useful during development, but be sure to remove any such code segments from your code before finishing your project. Masking comments should look like this:

```
//  if (moreRecs == true) {
//      ProcessRecord();
//  }
//  else {
//      doFileCleanUp();
//  }
```

General Tips About Comments

It is important to use comments where the code itself may not be clear, and it is equally important to avoid comments where the code is obvious. The following is a classic, from the Bad Comments Hall of Fame:

```
x = 5;           // set the variable x equal to 5
```

Comments should be used to provide summaries of complex code and to reveal information about the code that would otherwise be difficult to determine. Avoid comments that will fall out of date, i.e., write your comments as if they might have to last forever.

Declarations Are Fun

Declarations are a huge part of Java. They are also complex, loaded with rules, and if used sloppily can lead to bugs and poor maintainability. The following set of

guidelines is intended to make your code more readable, more debuggable, and more maintainable.

Sequencing Your Declarations

The elements in your Java source files should be arranged in a standard sequence. In some cases the compiler demands it, and for the rest of the cases consistency will help you win friends and influence people. Here goes:

- class comments
- package declaration
- import statements
- class declaration
- static variables
- instance variables
- constructors
- methods

Location and Initialization

The following guidelines should be considered when making Java declarations:

- Within methods:
  - Declare and initialize local variables before other statements (whenever possible).
  - Declare and initialize block variables before other block statements (when possible).
  - Declare only one member per line.
  - Avoid *shadowing* variables. This occurs when an instance variable has the same name as a local or block variable. While the compiler will allow it, *shadowing* is considered very unfriendly towards the next co-worker (remember: potentially psychopathic) who has to maintain your code.

Capitalization

Three guesses. You better use capitalization correctly when you declare and use your package, class, interface, method, variable, and constant names. The rules are pretty simple:

- **Package names** The safest bet is to use lowercase when possible:
  `com.wickedlysmart.utilities`
- **Class and Interface names** Typically they should be nouns; capitalize the first letter and any other first letters in secondary words within the name:
  `Customer or CustomTable`
- **Method names** Typically they should be verbs; the first word should be lowercase, and if there are secondary words, the first letter of each should be capitalized:
  `initialize(); or getTelescopicOrientation();`
- **Variable names** They should follow the same capitalization rules as methods; you should start them with a letter (even though you can use _ or $, don't), and only temporary variables like looping variables should use single character names:
  `currentIndex; or name; or x;`
- **Constant names** To be labeled a constant, a variable must be declared static and final. Their names should be all uppercase and underscores must be used to separate words:
  `MAX_HEIGHT; or USED;`

Key Points Summary

- This is the easiest part of the exam, if you are careful and thorough you should be able to do very well in this area.
- Always indent four spaces from the previous level of indentation.
- Break long lines at around the 65 character mark:
  - Break after a comma
  - Break before an operator

- Try not to break inside inner parens.
- Use single blank lines between constructors, methods, logic segments, before comments, and after your last instance variable.
- Use blank spaces between binary operators, after commas in argument lists, after *for* expressions, between a keyword and a paren.
- Place opening curly brace on the same line as the declaration or statement.
- Put closing curly brace on a new line.
- Closing curly brace shares a line with *else*, *else if*, *do*, *catch*, and *finally*.
- Block comments start and end with /* and */, * in the middle.
- Single line comments use /* */
- End of line comments use //
- Masking comments use //
- File declaration sequence is this: comments, package, import, class, static, instance, constructors, methods.
- Initialize variables at the top of blocks; avoid variable name shadowing.
- Package names are lowercase: com.wickedlysmart.utilities.
- Classes and interfaces have capitalized nouns for names: Inventory.
- Methods and variables names start lowercase and capitalize secondary words, as in
  `doRiskyStuff(); or currentIndex;.`
- Constant names are all caps with underscores: MAX_HEADROOM.