Hacettepe University
Department of Computer Engineering
BBM478 - Software Engineering Laboratory

Coding Standards

Group One

Ahmed Şamil BÜLBÜL (21426749, Software Developer)
Halil İbrahim ŞENER (21328447, Tester)
Naciye GÜZEL (21580841, Project Manager)

1. INTRODUCTION

1.1 Why Have Code Conventions

- It is important to create a code convention in order to increase teamwork between team members.
- Using same convention increases understanding of code, also readability.
- Maintaining is usually done by people who didn't participate coding process. This document will help them to understand original coding process.

1.2 Acknowledgments

This document reflects the Java language coding standards presented in the Java Language Specification, from Sun Microsystems. Major contributions are from Peter King, Patrick Naughton, Mike DeMoney, Jonni Kanerva, Kathy Walrath, and Scott Hommel.

For questions concerning adaptation, modification, or redistribution of this document, please read our copyright notice at http://java.sun.com/docs/codeconv/html/Copyright.doc.html.

Comments on this document should be submitted to our feedback form at http://java.sun.com/docs/forms/sendusmail.html.

2. FILE NAMES

2.1 File Suffix

LBLS uses the following suffixes:

File Type	Suffix
Java source	.java
Java bytecode	.class
Structured Query Language Data File	.sql

2.2 Common File Names

Frequently used file names include:

File Name	Use
POM	The file used to create Maven project and manage dependencies.
README	The preferred name for the file that summarizes the contents of a particular directory.

LICENSE	The preferred name for the file that summarizes how to contribute to LBLS project.

3. FILE ORGANIZATION

Files longer than 2000 lines are clumsy so should be avoided.

3.1 Java Source Files

Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class. The public class should be the first class or interface in the file.

It is optional that a class will have an explanation comments. Explanation comments gives summary of what class/method does.

4. INDENTATION

Four spaces should be used as the unit of indentation. The exact construction of the indentation (spaces vs. tabs) is unspecified.

4.1 Line Length

Avoid lines longer than 80 characters, since they're not handled well by many terminals and tools.

4.2 Wrapping Lines

When an expression will not fit on a single line, break it according to these general principles:

(Note: order goes from high-level brake to low-level breaks)

- Break after a comma.
- Break before an operator.
- Align the new line with the beginning of the expression at the same level on the previous line.
- For expressions more than two lines, lines after second one should be aligned with second line but with respect to rule above.
- Prefer higher-level breaks to lower-level breaks.

Break after a comma example:

```
method(veryLongExpressionParameter1, veryLongExpressionParameter2,
       veryLongExpressionParameter3, veryLongExpressionParameter4);
```

Break before an operator example:

```
variable = operand1 + operand2 * (operand3 % operand4) * operand5
           + operand6;
```

Following are two examples of indenting method declarations. The first is the conventional case. The second would shift the second and third lines to the far right if it used conventional indentation, so instead it indents only 8 spaces.

```
//CONVENTIONAL INDENTATION
```

```
someMethod(int anArg, Object anotherArg, String yetAnotherArg,
           Object andStillAnother) {
      ...
}

//INDENT 8 SPACES TO AVOID VERY DEEP INDENTS
private static synchronized horkingLongMethodName(int anArg,
        Object anotherArg, String yetAnotherArg,
        Object andStillAnother) {
      ...
}
```

5. COMMENTS

Java programs can have two kinds of comments: implementation comments and documentation comments. Implementation comments are those found in C++, which are delimited by `/*...*/`, and `//`. Documentation comments (known as "doc comments") are Java-only, and are delimited by `/**...*/`. Doc comments can be extracted to HTML files using the Javadoc tool.

Discussion of nontrivial or non obvious design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for redundant comments to get out of date. In general, avoid any comments that are likely to get out of date as the code evolves.

6. DECLARATIONS

6.1 Number Per Line

Even though one declaration per line is recommended, multiple declarations are also allowed whenever necessary.

```
String name, surname;
```

and

```
String name;
String surname;
```

are acceptable declarations.

6.2 Placement

Since creating views use Swing, declarations cannot be placed uniformly. In view classes, it is up to programmer. Except View classes, put declarations at the beginning of blocks as much as possible.

7. STATEMENTS

7.1 Simple Statements

Each line should contain at most one statement. What should not be done is:

```
var = method(exp1); var++;
```

7.2 Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces "`{ statements }`".

- The enclosed statements should be indented one more level than the compound statement.

```
if(var <= 0)
{
      var++;
      method(arg1);
}

if (arg == 0) {
      method(arg2);
}
```

8. WHITE SPACE

8.1 Blank Lines

Blank lines increases readability by dividing code sections by their relations.

- There should be at least one blank line between function definitions.
- There should be at least one blank line between class and interface definitions.

9. NAMING CONVENTIONS

It is important to have readable names throughout code to make it understandable. By doing this names can express themselves, their basic functionality and give information.

Identifier Type	**Naming Rules**	**Example**
Classes	Class names should be nouns, in mixed case with the first letter of each internal word capitalized. Try to keep your class names simple and descriptive. Use whole words - avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).	`class BookController`
Interfaces	Interface names should be capitalized like class names.	`interface RasterDelegate`
Methods	Methods should be verbs, in mixed case with the first letter lowercase, with the first letter of each internal word capitalized.	`run();` `printToFile();`
Variables	Except for variables, all instance, class, and class constants are in	`int i;` `char c;`

	mixed case with a lower-case first letter. Internal words start with capital letters. Variable names should be short yet meaningful. The choice of a variable name should be mnemonic - that is, designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.	`CarDriver myDriver;`
Constants	The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by under-scores ("_"). (ANSI constants should be avoided, for ease of debugging.)	`int MIN_VALUE = 0;`