Naming and Style Guidelines for Java

School of Engineering and Technology, UWT *Fall 2020*

1 Introduction

Code standards aren't just about obsession; they are about productivity, professionalism, and presentation. Code is more often read than written and products that adhere to code standards are more readable, maintainable, robust, testable, and professional than those that do not.

The general guidelines you should follow for Java code are shown at http://web.archive.org/web/20140222045352/http://www.oracle.com/technetwork/java/codeconv-138413.html .
Other sources for general programming standards and for specific standards for Java are KindSoftware Coding Standards at http://www.kindsoftware.com/documents/whitepapers/code_standards
and Google's Java coding style guide at https://google.github.io/styleguide/javaguide.html
The material from these resources is not reproduced here. This document details the specific naming and style guidelines for Java entities that are enforced by CheckStyle templates in various courses at UWT and elsewhere.

Note that there is no "one true way" of formatting Java code (or, indeed, any code). You should probably distrust anybody who tells you there is. However, the particular set of guidelines presented here have proven useful for a number of reasons, not the least of which is that they enforce a discipline in coding that will serve you well later regardless of what code standard you are asked to follow.

2 Tabs, Spacing and Braces

The "tab" character (\t) should not appear in any source file, unless it is part of a string literal (that is, you define a string that contains a "\t" in it for user interaction or other purposes). Rationale: tab characters lead to inconsistent code appearance in different editors/IDEs; in addition, because tabs are invisible, it is impossible for you to tell whether what you have at the beginning of a line is (for example) 8 spaces or 2 tab characters.

The standard indentation width is 4 spaces per indentation level. Each nested block of your code should appear as a different indentation level. Rationale: it is the most commonly used standard indentation width and gives a nice balance between use of horizontal space and logical delineation of code blocks. Since typing 4 spaces each time you need to indent a line is quite cumbersome, you may want to change your editor settings to replace tabs with 4 blank spaces.

Spacing is enforced between operators and operands (e.g., “a + b” rather than “a+b”), between commas or semicolons and entities that follow them (e.g., “int theNumber, char theLetter” rather than “int theNumber,char theLetter”), and in a number of other places. The exception is parentheses. A keyword followed by a parenthesis should be separated by a space unless it is a method name (e.g. “while (true)” and “public void someMethod()”). However, there should be no blank spaces between the opening parenthesis and the first entity that follows or between the last entity inside parentheses and the closing parenthesis. Rationale: this spacing generally makes identifiers easier to see, and code therefore easier to read.

Brace placement can be done in either of the two commonly-accepted ways: with the opening brace on the same line as a declaration/statement (Figure 1) or with the opening brace on the next line (Figure 2)

```
public void m() {
    if (myFlag) {
        // do something
    } else {
        // do something else
    }
}
```

Figure 1: The “same line” method of brace placement.

```
public void m()
{
    if (myFlag)
    {
        // do something
    }
    else
    {
        // do something else
    }
}
```

Figure 2: The “next line” method of brace placement.

In each homework project, only one of these two styles may be used. Rationale: consistency makes code easier to comprehend.

CheckStyle files and Eclipse format files are provided for both brace styles.

3 Usage of the `final` Modifier

The Java `final` modifier must be used on every method parameter (unconditionally), as well as on every field and local variable that will, in fact, remain unchanged during execution. The FindBugs and PMD templates will warn you if the `final` modifier has been omitted from a field that appears final. **Important:** Because the tools cannot predict how you will use fields and local variables in code you have not written yet, you may see a lot of `final`-related warnings early. You can safely ignore warnings about non-`final` fields and local variables when you have not yet written all the code that uses those fields and local variables.

Rationale: by marking method parameters `final`, you completely eliminate bugs that result from accidentally assigning values to them (something that is almost never intentional, and is always unintuitive). Also, every `final` field and local variable is one that you can effectively ignore when debugging your code.

4 Class, Method, Field, Parameter and Variable Naming

The following rules apply to the naming of various Java entities. In all cases, field names should be as self-documenting as possible. This may involve making them longer than you're used to; that's OK, since modern IDEs have auto-completion and modern compilers and runtime environments don't care about identifier lengths.

- Classes and interfaces are named InCamelCase, starting with an uppercase letter, that is, according to standard Java conventions.
- Methods are named inCamelCase, starting with a lowercase letter, that is, according to standard Java conventions.
- `static final` fields (constants) are named `IN_ALL_CAPS_WITH_UNDERSCORES`; that is, according to standard Java convention. However, their names may not start with `MY_`, `THE_`, or any of the other "special" prefixes described below.
- Instance fields are named like `myFieldName`; that is, starting with `my` and using camel case thereafter.
- Method parameters are named `theName` or `thing1/2/3`; that is, starting with `the` and using camel case thereafter, or starting in lower case and ending with a digit.
- Local variables are named `inCamelCase`; that is, just like instance fields, but with no leading `my` or `the` to differentiate them from fields and method parameters.

Rationale: There are several reasons for this naming scheme. First, it prevents you from accidentally shadowing variable names since local variables, method parameters, and instance fields cannot have identical names. Second, it makes it obvious when you are manipulating object state (`my`, indicating that a field "belongs to" its containing object) or static state (`CAPS`), which can affect later method calls or other threads, rather than local state, which cannot affect anything other than the current thread. Finally, it makes object-oriented analysis and debugging easier (you can immediately identify what kind of state an identifier represents).

In summary, if I have a variable to represent a "time stamp", for instance, I would name it differently depending on where and how it is used in the project:

when used as a class constant – `TIME_STAMP`
when used as an instance field – `myTimeStamp`
when used as a method parameter – `theTimeStamp` or `timeStamp1`
when used a local variable in a method - `timeStamp`

5 Documentation

Every class, method, and field—not just the `public` ones—must have *complete* Javadoc documentation. For more information on Javadoc, see the Oracle tutorial on writing Javadoc comments at http://java.sun.com/j2se/javadoc/writingdoccomments/index.html

In addition to full Javadoc, every source file should have a non-Javadoc header comment identifying it as belonging to a specific project/fileset (this is where copyright/licensing information would go in a publicly-released software project,[1] and where general information about the assignment should go on homework). For example, the code shown in Figure 3 would be a reasonable start for a class called `Example` (this also obeys the import rules described in section 6).

Rationale: Everything has to be documented somehow. Having a consistent header comment across all files in a project makes it easy to identify files as part of that project, and provides a single point of reference for copyright/license information. Using Javadoc for all in-code documentation enforces consistency among that documentation and reduces the amount of time you have to spend deciding whether entities need Javadoc comments. It also means that you don't have to rewrite/modify comments when you change protection levels of entities in your code during refactoring.

[1]Note that you actually own the copyright on all code you write for class assignments and projects, and you might want to indicate so explicitly in your file headers.

```
/*
 *  Java Naming and Style Guidelines
 *  Fall 2020
 */

package mypackage;

import java.util.List;
import java.util.Set;
import javax.swing.JFrame;
import mypackage.util.UsefulClass;

/**
 *  Example is a class that conforms to the Naming and
 *  Style Guidelines for Java you are reading right now.
 *
 *  @author  Your Name
 *  @version  Day Month Year
 */
public class Example
{
  // code for class Example goes here
}
```

Figure 3: A conforming source file, including imports and class Javadoc but no code.

6 Import and Package Organization

Import and package statements should be organized for readability, according to the following rules. Most IDEs will do this organization for you automatically.

- The package statement, if any, appears immediately after the header comment of each source file and before any import statements.
- All import statements appear after the package statement (or header comment, in the absence of a package statement) of each source file and before the class Javadoc comment.
- Groups of imports are separated by a single blank line.
- Imports are listed in alphabetical order within their groups.

- Imports from `java.` packages, if any, are declared in the first group.
- Imports from `javax.` packages, if any, are declared in the next group.
- Other imports are declared in the next group (you can optionally make multiple groups of these, as you see fit).

7 Code Organization

Code should be organized according to information hiding and `static` modifiers as follows (enforced by CheckStyle):

- All fields are declared before all constructors.
- All constructors are declared before all other methods.
- All `static` fields are declared before all non-`static` fields.
- All `static` methods are declared before all non-`static` methods.
- All inner and nested classes are declared after all fields and methods.
- Among the same type of entity (constructors, instance methods, static methods, instance fields, static fields), the order for information hiding modifiers should be `public`, then `protected`, then package/default, then `private`.

8 Miscellaneous

The following typical coding issues are automatically detected by the tool configurations that accompany this standard:

- Too many exit points from methods. A method should typically have only one exit point (that is, zero or one `return` statements). The exception to this rule is `equals` methods, where it is very common to short-circuit the return value.
- Deeply nested `if` statements, or an overabundance of `if/case` statements in a single method. In general, this means that your logic needs to be reworked into something simpler and perhaps broken into multiple methods.
- Exceptionally long constructors and methods.

- Exceptionally long classes.
- Constants used outside of `static final` declarations ("magic numbers" or "magic strings"). In general, you should declare symbolic constants when necessary (*e.g.* `public static final int NUM_PRIMARY_COLORS = 3`). However, you should *never* declare symbolic constants of the form "`public static final int THREE = 3`" to appease the static checking system; such constants do not make the code any easier to understand and will definitely be caught by your instructor.

The static checkers detect many other coding issues as well; in some cases, their suggestions can be ignored, but in most cases you will want to follow them. Always ask your instructor if you have a question about a particular message generated by a static checker! Remember, the static checkers are there to help you write better code.