CST8110 Java Style Guide

Introduction

This document serves as the complete definition of CST8110's coding standards for source code in the Java Programming Language. A Java source file is described as being in good style for CST8110 if and only if it adheres to the rules herein.

Terminology Notes

In this document, unless otherwise clarified:

1. The term **class** is used inclusively to mean an "ordinary" class, enum class, interface or annotation type (@interface).
2. The term **member** (of a class) is used inclusively to mean a nested class, field, method, or constructor; that is, all top-level contents of a class except initializers and comments.
3. The term **comment** always refers to implementation comments. We do not use the phrase "documentation comments", instead using the common term "Javadoc."

Other "terminology notes" may appear occasionally throughout the document.

Source file basics

1. **File name**
    a. The source file name consists of the case-sensitive name of the top-level class it contains (of which there is exactly one), plus the `.java` extension.
2. **File encoding: UTF-8**
    a. Source files are encoded in UTF-8.
3. **Special characters**
    a. Whitespace characters
        i. Aside from the line terminator sequence, the ASCII horizontal space character (0x20) is the only whitespace character that appears anywhere in a source file. This implies that:
            - All other whitespace characters in string and character literals are escaped.
            - Tab characters are not used for indentation.
    b. Special escape sequences
        i. For any character that has a special escape sequence (\b, \t, \n, \f, \r, \", \' and \\), that sequence is used rather than the corresponding octal (e.g. \012) or Unicode (e.g. \u000a) escape.

Source file structure

A source file consists of, **in order**:

1. File header

2. Import statements
3. Exactly one top-level class

Exactly one blank line separates each section that is present.

1. **File header**

```
// Algonquin College CST8110
// Name: <Your Name>
// Student ID: <Your ID>
// Section: <Lab Section>
// Assignment: Lab <#>
// Due Date: <YYYY-MM-DD>
```

Example:

```
// Algonquin College CST8110
// Name: Angela Giddings
// Student ID: 045512345
// Section: 311
// Assignment: Lab 7
// Due Date: 2019-01-31
```

2. **Package statement**
    a. The package statement is **not line-wrapped**. The column limit (Column limit: 100) does not apply to package statements.
3. **Import statements**
    a. No wildcard imports
        i. **Wildcard imports**, static or otherwise, **are not used**.
    b. No line-wrapping
        i. Import statements are **not line-wrapped**. The column limit (Column limit: 100) does not apply to import statements.
    c. Ordering and spacing
        i. Imports are ordered as follows:
            1. All static imports in a single block.
            2. All non-static imports in a single block.
        ii. If there are both static and non-static imports, a single blank line separates the two blocks. There are no other blank lines between import statements.
        iii. Within each block the imported names appear in ASCII sort order. (**Note**: this is not the same as the import statements being in ASCII sort order, since '.' sorts before ';'.)
4. **Class declaration**
    a. Exactly one top-level class declaration

i. Each top-level class resides in a source file of its own.

5. **Ordering of class contents**
    a. The order you choose for the members and initializers of your class can have a great effect on learnability. However, there's no single correct recipe for how to do it; different classes may order their contents in different ways.
    b. What is important is that each class uses ***some* logical order**, which its maintainer could explain if asked. For example, new methods are not just habitually added to the end of the class, as that would yield "chronological by date added" ordering, which is not a logical ordering.
    c. Overloads: never split
        i. When a class has multiple constructors, or multiple methods with the same name, these appear sequentially, with no other code in between (not even private members).

Formatting

Terminology Note: block-like construct refers to the body of a class, method or constructor. Note that any array initializer may optionally be treated as if it were a block-like construct.

1. **Braces**
    a. Braces are used where optional
        i. Braces are used with `if`, `else`, `for`, `do` and `while` statements, even when the body is empty or contains only a single statement.
    b. Nonempty blocks: K & R style
        i. Braces follow the Kernighan and Ritchie style ("Egyptian brackets") for *nonempty* blocks and block-like constructs:
            - No line break before the opening brace.
            - Line break after the opening brace.
            - Line break before the closing brace.
            - Line break after the closing brace, *only if* that brace terminates a statement or terminates the body of a method, constructor, or *named* class. For example, there is *no* line break after the brace if it is followed by `else` or a comma.
        ii. Examples:

```
return () -> {
  while (condition()) {
    method();
  }
};

return new MyClass() {
  @Override public void method() {
    if (condition()) {
```

```
      try {
        something();
      } catch (ProblemException e) {
        recover();
      }
    } else if (otherCondition()) {
      somethingElse();
    } else {
      lastThing();
    }
  }
};
```

c. Empty blocks: may be concise
   i. An empty block or block-like construct may be in K & R style. Alternatively, it may be closed immediately after it is opened, with no characters or line break in between ({}), **unless** it is part of a *multi-block statement* (one that directly contains multiple blocks: `if/else` or `try/catch/finally`).
   ii. Examples:

```
  // This is acceptable
  void doNothing() {}

  // This is equally acceptable
  void doNothingElse() {
  }
```

```
  // This is not acceptable: No concise empty blocks in a multi-block statement
  try {
    doSomething();
  } catch (Exception e) {}
```

2. **Block indentation: +2 spaces**
   a. Each time a new block or block-like construct is opened, the indent increases by two spaces. When the block ends, the indent returns to the previous indent level. The indent level applies to both code and comments throughout the block.
3. **One statement per line**
   a. Each statement is followed by a line break.
4. **Column limit: 100**
   a. Java code has a column limit of 100 characters. A "character" means any Unicode code point. Except as noted below, any line that would exceed this limit must be line-wrapped.

b. Each Unicode code point counts as one character, even if its display width is greater or less. For example, if using fullwidth characters, you may choose to wrap the line earlier than where this rule strictly requires.

c. Exceptions:

i. Lines where obeying the column limit is not possible (for example, a long URL in Javadoc, or a long JSNI method reference).

ii. `package` and `import` statements.

iii. Command lines in a comment that may be cut-and-pasted into a shell.

5. Line-wrapping

Terminology Note: When code that might otherwise legally occupy a single line is divided into multiple lines, this activity is called line-wrapping.

There is no comprehensive, deterministic formula showing exactly how to line-wrap in every situation. Very often there are several valid ways to line-wrap the same piece of code.

Note: While the typical reason for line-wrapping is to avoid overflowing the column limit, even code that would in fact fit within the column limit may be line-wrapped at the author's discretion.

Tip: Extracting a method or local variable may solve the problem without the need to line-wrap.

a. Where to break

The prime directive of line-wrapping is: prefer to break at a **higher syntactic level**. Also:

- When a line is broken at a *non-assignment* operator the break comes *before* the symbol.
- This also applies to the following "operator-like" symbols, such as the dot separator (`.`)
- When a line is broken *at* an assignment operator the break typically comes *after* the symbol, but either way is acceptable.
  - This also applies to the "assignment-operator-like" colon in an enhanced `for` ("foreach") statement.
- A method or constructor name stays attached to the open parenthesis (`(`) that follows it.
- A comma (`,`) stays attached to the token that precedes it.

Note: The primary goal for line wrapping is to have clear code, *not necessarily* code that fits in the smallest number of lines.

b. Indent continuation lines at least +4 spaces

i. When line-wrapping, each line after the first (each *continuation line*) is indented at least +4 from the original line.
ii. When there are multiple continuation lines, indentation may be varied beyond +4 as desired. In general, two continuation lines use the same indentation level if and only if they begin with syntactically parallel elements.
iii. The section on Horizontal alignment addresses the discouraged practice of using a variable number of spaces to align certain tokens with previous lines.

6. Whitespace

a. Vertical whitespace

i. A single blank line always appears:

- *Between* consecutive members or initializers of a class: fields, constructors, methods, nested classes, static initializers, and instance initializers.
    a. **Exception**: A blank line between two consecutive fields (having no other code between them) is optional. Such blank lines are used as needed to create *logical groupings* of fields.
- As required by other sections of this document (such as Source file structure, and, Import statements).

ii. A single blank line may also appear anywhere it improves readability, for example between statements to organize the code into logical subsections. A blank line before the first member or initializer, or after the last member or initializer of the class, is neither encouraged nor discouraged.
iii. *Multiple* consecutive blank lines are permitted, but never required (or encouraged).

b. Horizontal whitespace

i. Beyond where required by the language or other style rules, and apart from literals, comments and Javadoc, a single ASCII space also appears in the following places **only**.

- Separating any reserved word, such as `if`, `for` or `catch`, from an open parenthesis (`(`) that follows it on that line
- Separating any reserved word, such as `else` or `catch`, from a closing curly brace (`}`) that precedes it on that line
- Before any open curly brace (`{`), with two exceptions:
    a. `@SomeAnnotation({a, b})` (no space is used)
    b. `String[][] x = {{"foo"}};`
- On both sides of any binary or ternary operator. This also applies to the following "operator-like" symbols:

a. the colon (`:`) in an enhanced `for` ("foreach") statement

but not

b. the dot separator (`.`), which is written like `object.toString()`

- After `,` `:` `;` or the closing parenthesis (`)`) of a cast
- On both sides of the double slash (`//`) that begins an end-of-line comment. Here, multiple spaces are allowed, but not required.
- Optional just inside both braces of an array initializer
  a. `new int[] {5, 6}` and `new int[] { 5, 6 }` are both valid
- Between a type annotation and `[]` or ....

This rule is never interpreted as requiring or forbidding additional space at the start or end of a line; it addresses only *interior* space.

c. Horizontal alignment: never required

Terminology Note: *Horizontal alignment* is the practice of adding a variable number of additional spaces in your code with the goal of making certain tokens appear directly below certain other tokens on previous lines.

This practice is permitted, but is never required for CST8110. It is not even required to *maintain* horizontal alignment in places where it was already used.

Here is an example without alignment, then using alignment:

```
private int x; // this is fine
private Color color; // this too

private int   x;      // permitted, but future edits
private Color color;  // may leave it unaligned
```

Tip: Alignment can aid readability, but it creates problems for future maintenance. Consider a future change that needs to touch just one line. This change may leave the formerly-pleasing formatting mangled, and that is **allowed**. More often it prompts the coder (perhaps you) to adjust whitespace on nearby lines as well, possibly triggering a cascading series of reformattings. That one-line change now has a "blast radius." This can at worst result in pointless busywork, but at best it still corrupts version history information, slows down reviewers and exacerbates merge conflicts.

7. **Grouping parentheses: recommended**
    a. Optional grouping parentheses are omitted only when author and reviewer agree that there is no reasonable chance the code will be misinterpreted without them, nor would they have made the code easier to read. It is *not* reasonable to assume that every reader has the entire Java operator precedence table memorized.
8. **Specific constructs**
    a. **Variable declarations**
        i. **One variable per declaration**
            - Every variable declaration (field or local) declares only one variable: declarations such as `int a, b;` are not used.
            - **Exception:** Multiple variable declarations are acceptable in the header of a `for` loop.
        ii. **Declared when needed**
            - Local variables are not habitually declared at the start of their containing block or block-like construct. Instead, local variables are declared close to the point they are first used (within reason), to minimize their scope. Local variable declarations typically have initializers, or are initialized immediately after declaration.
    b. **Arrays**
        i. **Array initializers: can be "block-like"**
        Any array initializer may *optionally* be formatted as if it were a "block-like construct." For example, the following are all valid (**not** an exhaustive list):

```
new int[] {           new int[] {
  0, 1, 2, 3            0,
}                       1,
                        2,
new int[] {             3,
  0, 1,                 }
  2, 3
}                     new int[]
                          {0, 1, 2, 3}
```

    c. **No C-style array declarations**
        i. The square brackets form a part of the *type*, not the variable: `String[] args`, **not** `String args[].`
    d. **Switch statements**
    Terminology Note: Inside the braces of a *switch block* are one or more *statement groups*. Each statement group consists of one or more *switch labels* (either `case FOO:` or `default:`), followed by one or more statements (or, for the *last* statement group, *zero* or more statements).

i. **Indentation**
   - As with any other block, the contents of a switch block are indented +2.
   - After a switch label, there is a line break, and the indentation level is increased +2, exactly as if a block were being opened. The following switch label returns to the previous indentation level, as if a block had been closed.

ii. **Fall-through: commented**
   - Within a switch block, each statement group either terminates abruptly (with a `break`, `continue`, `return` or thrown exception), or is marked with a comment to indicate that execution will or *might* continue into the next statement group. Any comment that communicates the idea of fall-through is sufficient (typically `// fall through`). This special comment is not required in the last statement group of the switch block. Example:

```
switch (input) {
  case 1:
  case 2:
    prepareOneOrTwo();
    // fall through
  case 3:
    handleOneTwoOrThree();
    break;
  default:
    handleLargeNumber(input);
}
```

Notice that no comment is needed after `case 1:`, only at the end of the statement group.

iii. **The `default` case is present**
   - Each switch statement includes a `default` statement group, even if it contains no code.

e. **Comments**

This section addresses *implementation comments*. Javadoc is addressed separately.

Any line break may be preceded by arbitrary whitespace followed by an implementation comment. Such a comment renders the line non-blank.

i. **Block comment style**
   - Block comments are indented at the same level as the surrounding code. They may be in `/* ... */` style or `// ...`

style. For multi-line `/* ... */` comments, subsequent lines must start with `*` aligned with the `*` on the previous line.

```
/*
 * This is          // And so           /* Or you can
 * okay.            // is this.          * even do this. */
 */
```

Comments are not enclosed in boxes drawn with asterisks or other characters.

Tip: When writing multi-line comments, use the `/* ... */` style if you want automatic code formatters to re-wrap the lines when necessary (paragraph-style). Most formatters don't re-wrap lines in `// ...` style comment blocks.

f. Modifiers

Class and member modifiers, when present, appear in the order recommended by the Java Language Specification:

```
public protected private abstract default static final transient volatile
synchronized native strictfp
```

g. Numeric Literals

long-valued integer literals use an uppercase `L` suffix, never lowercase (to avoid confusion with the digit `1`). For example, `3000000000L` rather than `3000000000l`.

Naming

1. **Rules common to all identifiers**

   Identifiers use only ASCII letters and digits, and, in a small number of cases noted below, underscores. Thus each valid identifier name is matched by the regular expression `\w+ .`

   In CST8110 Style, special prefixes or suffixes are not used. For example, these names are **not** CST8110 Style: `name_`, `mName`, `s_name` and `kName`.

2. **Rules by identifier type**

   a. **Class names**

      i. Class names are written in UpperCamelCase.

      ii. Class names are typically nouns or noun phrases. For example, `Character` or `ImmutableList`. Interface names may also be nouns or noun phrases (for example, `List`), but may sometimes be adjectives or adjective phrases instead (for example, `Readable`).

      iii. There are no specific rules or even well-established conventions for naming annotation types.

iv. Test classes are named starting with the name of the class they are testing, and ending with Test. For example, `HashTest` or `HashIntegrationTest`.

b. Method names

i. Method names are written in lowerCamelCase.

ii. Method names are `typically` verbs or verb phrases. For example, `sendMessage` or stop.

iii. Underscores may appear in JUnit test method names to separate logical components of the name, with each component written in lowerCamelCase. One typical pattern is `<methodUnderTest>_<state>`, for example `pop_emptyStack`. There is no One Correct Way to name test methods.

c. Constant names

i. Constant names use `CONSTANT_CASE`: all uppercase letters, with each word separated from the next by a single underscore. But what *is* a constant, exactly?

1. Constants are static final fields whose contents are deeply immutable and whose methods have no detectable side effects. This includes primitives, Strings, immutable types, and immutable collections of immutable types. If any of the instance's observable state can change, it is not a constant. Merely *intending* to never mutate the object is not enough. Examples:

```
// Constants
static final int NUMBER = 5;
static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");
static final ImmutableMap<String, Integer> AGES = ImmutableMap.of("Ed", 35, "Ann",
32);
static final Joiner COMMA_JOINER = Joiner.on(','); // because Joiner is immutable
static final SomeMutableType[] EMPTY_ARRAY = {};
enum SomeEnum { ENUM_CONSTANT }
```

```
// Not constants
static String nonFinal = "non-final";
final String nonStatic = "non-static";
static final Set<String> mutableCollection = new HashSet<String>();
static final ImmutableSet<SomeMutableType> mutableElements =
ImmutableSet.of(mutable);
static final ImmutableMap<String, SomeMutableType> mutableValues =
    ImmutableMap.of("Ed", mutableInstance, "Ann", mutableInstance2);
static final Logger logger = Logger.getLogger(MyClass.getName());
static final String[] nonEmptyArray = {"these", "can", "change"};
```

ii. These names are typically nouns or noun phrases.

d. Non-constant field names

i. Non-constant field names (static or otherwise) are written in lowerCamelCase.

ii. These names are typically nouns or noun phrases. For example, `computedValues` or `index`.

e. Parameter names

i. Parameter names are written in lowerCamelCase.

ii. One-character parameter names in public methods should be avoided.

f. Local variable names

i. Local variable names are written in lowerCamelCase.

ii. Even when final and immutable, local variables are not considered to be constants, and should not be styled as constants.

3. Camel case: defined

Sometimes there is more than one reasonable way to convert an English phrase into camel case, such as when acronyms or unusual constructs like "IPv6" or "iOS" are present. To improve predictability, CST8110 Style specifies the following (nearly) deterministic scheme.

Beginning with the prose form of the name:

- Convert the phrase to plain ASCII and remove any apostrophes. For example, "Müller's algorithm" might become "Muellers algorithm".
- Divide this result into words, splitting on spaces and any remaining punctuation (typically hyphens).
    - Recommended: if any word already has a conventional camel-case appearance in common usage, split this into its constituent parts (e.g., "AdWords" becomes "ad words"). Note that a word such as "iOS" is not really in camel case per se; it defies any convention, so this recommendation does not apply.
- Now lowercase everything (including acronyms), then uppercase only the first character of:
    - ... each word, to yield upper camel case, or
    - ... each word except the first, to yield lower camel case
- Finally, join all the words into a single identifier.

Note that the casing of the original words is almost entirely disregarded.
Examples:

Prose Form	Correct	Incorrect
XML HTTP request	XmlHttpRequest	XMLHTTPRequest

new customer ID	newCustomerId	newCustomerID
inner stopwatch	innerStopwatch	innerStopWatch
supports IPv6 on iOS?	supportsIpv6OnIos	supportsIPv6OnIOS
YouTube importer	YouTubeImporter	YoutubeImporter*

*Acceptable, but not recommended.

Note: Some words are ambiguously hyphenated in the English language: for example "nonempty" and "non-empty" are both correct, so the method names `checkNonempty` and `checkNonEmpty` are likewise both correct.

Programming Practices

1. **Static members: qualified using class**
    a. When a reference to a static class member must be qualified, it is qualified with that class's name, not with a reference or expression of that class's type.

```
Foo aFoo = ...;
Foo.aStaticMethod(); // good
aFoo.aStaticMethod(); // bad
somethingThatYieldsAFoo().aStaticMethod(); // very bad
```

Javadoc

1. **Formatting**
    a. **General form**

The basic formatting of Javadoc blocks is as seen in this example:

```
/**
 * Multiple lines of Javadoc text are written here,
 * wrapped normally...
 */
public int method(String p1) { ... }
```

... or in this single-line example:

```
/** An especially short bit of Javadoc. */
```

The basic form is always acceptable. The single-line form may be substituted when the entirety of the Javadoc block (including comment markers) can fit on a single line. Note that this only applies when there are no block tags such as `@return`.

b. **Paragraphs**

One blank line—that is, a line containing only the aligned leading asterisk (*)—appears between paragraphs, and before the group of block tags if present. Each paragraph but the first has `<p>` immediately before the first word, with no space after.

c. **Block tags**

Any of the standard "block tags" that are used appear in the order `@param`, `@return`, `@throws`, `@deprecated`, and these four types never appear with an empty description. When a block tag doesn't fit on a single line, continuation lines are indented four (or more) spaces from the position of the `@`.

2. **The summary fragment**

Each Javadoc block begins with a brief summary fragment. This fragment is very important: it is the only part of the text that appears in certain contexts such as class and method indexes.

This is a fragment—a noun phrase or verb phrase, not a complete sentence. It does not begin with `A {@code Foo} is a...`, or `This method returns...`, nor does it form a complete imperative sentence like `Save the record.`. However, the fragment is capitalized and punctuated as if it were a complete sentence.

Tip: A common mistake is to write simple Javadoc in the form `/** @return the customer ID */`. This is incorrect, and should be changed to `/** Returns the customer ID. */`.

3. **Where Javadoc is used**

At the *minimum*, Javadoc is present for every `public` class, and every `public` or `protected` member of such a class, with a few exceptions noted below.

Additional Javadoc content may also be present.

4. **Non-required Javadoc**

Other classes and members have Javadoc as needed or desired.

Whenever an implementation comment would be used to define the overall purpose or behavior of a class or member, that comment is written as Javadoc instead (using `/**`).