Coding Conventions

The following style guide describes the coding conventions used for the source code of the jTNC library and the demonstrator of a smart metering system. For the creation of this style guide, the Google Java Style was used as basis. The Google Java Style can be found at https://code.google.com/p/google-styleguide/ and is licensed under the CC BY 3.0 License[1]. The changes that are made to adjust the Google Style Guide are licensed under the CC-BY 4.0 License[2], while the unchanged parts remain licensed as parts of the original Google Style Guide.

1   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 *comment* always refers to *implementation* comments. We do not use the phrase "documentation comments", instead using the common term "Javadoc."

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

2 Source file basics

2.1 File name

The source file name consists of the case-sensitive name of the top-level class it contains, plus the `.java` extension.

2.2 File encoding: UTF-8

Source files are encoded in UTF-8.

2.3 Whitespace characters

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:

1. All other whitespace characters in string and character literals are escaped.

2. Tab characters are not used for indentation.

2.3.1   Special escape sequences

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.

2.3.2 Non-ASCII characters

For the remaining non-ASCII characters, either the actual Unicode character (e.g. ∞) or the equivalent Unicode escape (e.g. \u221e) is used, depending only on which makes the code easier to read and understand.

Tip: In the Unicode escape case, and occasionally even when actual Unicode characters are used, an explanatory comment can be very helpful.

3 Source file structure

A source file consists of, in order:

1. License or copyright information, if present
2. Import statements
3. Exactly one top-level class

3.1 License or copyright information, if present

If license or copyright information belongs in a file, it belongs here.

3.2 Import statements

3.2.1 No wildcard imports

Wildcard imports, static or otherwise, are not used.

3.2.2 Ordering and spacing

Import statements are divided into groups by top level package. Each groups is separated by a single blank line.

3.3 Class declaration

3.3.1 Exactly one top-level class declaration

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

3.3.2 Class member ordering

The ordering of the members of a class can have a great effect on learnability, but there is no single correct recipe for how to do it. Different classes may order their members differently.

What is important is that each class order its members in *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.

3.3.3 Overloads: never split

When a class has multiple constructors, or multiple methods with the same name, these appear sequentially, with no intervening members.

3.4 package-info.java

Every Java package contains a package-info.java file. The file does not contain a class. Instead it must consist of the following in order:

1. License or copyright information, if present
2. Package statement
3. Import statements

The conventions for formatting and documentation and so on with in this document apply to the listed elements as to the source files that contain a Java class.

4 Formatting

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

4.1 Braces

4.1.1 Braces are used where optional

Braces are used with `if`, `else`, `for`, `do` and `while` statements, even when the body is empty or contains only a single statement.

4.1.2 Nonempty blocks: K & R style

Braces follow the Kernighan and Ritchie style ("Egyptian brackets[3]") 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 *if* that brace terminates a statement or 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.

3http://www.codinghorror.com/blog/2012/07/new-programming-jargon.html

Example:

```
return new MyClass() {
    @Override
    public void method() {
        if (condition()) {
            try {
                something();
            } catch (ProblemException e) {
                recover();
            }
        }
    }
};
```

A few exceptions for enum classes are given in the section about Enum classes.

4.1.3 Empty blocks: may be concise

An empty block or block-like construct *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-if/else or try/catch/finally).

Example:

```
    void doNothing() {}
```

4.2 Block indentation: +4 spaces

Each time a new block or block-like construct is opened, the indent increases by four 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. (See the example in the section about Nonempty blocks: K & R Style.)

4.3 One statement per line

Each statement is followed by a line-break.

4.4 Column limit: 80

Projects have to use a column limit of 80 characters. Except as noted below, any line that would exceed this limit must be line-wrapped, as explained in the section Line-wrapping.

Exceptions:

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

4.5 Line-wrapping

Terminology Note: When code that might otherwise legally occupy a single line is divided into multiple lines, typically to avoid overflowing the column limit, 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.

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

4.5.1 Where to break

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

1. When a line is broken at a *non-assignment* operator the break comes *before* the symbol.
    o This also applies to the following "operator-like" symbols: the dot separator (`.`), the ampersand in type bounds (`<T extends Foo & Bar>`), and the pipe in catch blocks (`catch (FooException | BarException e)`).

2. When a line is broken at an *assignment* operator the break typically comes *after* the symbol, but either way is acceptable.
    o This also applies to the "assignment-operator-like" colon in an enhanced `for` ("foreach") statement.

3. A method or constructor name stays attached to the open parenthesis (`(`) that follows it.

4. A comma (`,`) stays attached to the token that precedes it.

4.5.2 Indent continuation lines at least +4 spaces

When line-wrapping, each line after the first (each *continuation line*) is indented at least +2 from the original line.

When there are multiple continuation lines, indentation may be varied beyond +2 as desired. In general, two continuation lines use the same indentation level if and only if they begin with syntactically parallel elements.

The section on Horizontal alignment addresses the discouraged practice of using a variable number of spaces to align certain tokens with previous lines.

4.6 Whitespace

4.6.1 Vertical Whitespace

A single blank line appears:

1. *Between* consecutive members (or initializers) of a class: fields, constructors, methods, nested classes, static initializers, instance initializers.
    o 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.

2. Within method bodies, as needed to create *logical groupings* of statements.

3. *Optionally* before the first member or after the last member of the class (neither encouraged nor discouraged).

4. As required by other sections of this document (such as the section about Import statements).

Multiple consecutive blank lines are permitted, but never required (or encouraged).

4.6.2   Horizontal whitespace

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.

1. Separating any reserved word, such as `if`, `for` or `catch`, from an open parenthesis (`(`) that follows it on that line

2. Separating any reserved word, such as `else` or `catch`, from a closing curly brace (`}`) that precedes it on that line

3. Before any open curly brace (`{`), with two exceptions:

    o   `@SomeAnnotation({a, b})` (no space is used)

    o   `String[][] x = {{"foo"}};` (no space is required between `{{`, by item 8 below)

4. On both sides of any binary or ternary operator. This also applies to the following "operator-like" symbols:

    o   the ampersand in a conjunctive type bound: `<T extends Foo & Bar>`

    o   the pipe for a catch block that handles multiple exceptions: `catch (FooException | BarException e)`

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

5. After `,`:`;` or the closing parenthesis (`)`) of a cast

6. On both sides of the double slash (`//`) that begins an end-of-line comment. Here, multiple spaces are allowed, but not required.

7. Between the type and variable of a declaration: List<String> `list`

8. *Optional* just inside both braces of an array initializer

    o   `new int[] {5, 6}` and `new int[] { 5, 6 }` are both valid

Note: This rule never requires or forbids additional space at the start or end of a line, only *interior* space.

4.6.3   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 by this style guide. 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.

4.7 Grouping parentheses: recommended

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.

4.8 Specific constructs

4.8.1   Enum classes

After each comma that follows an enum constant, a line-break is optional.

An enum class with no methods and no documentation on its constants may optionally be formatted as if it were an array initializer (see section on array initializers).

```
private enum Suit { CLUBS, HEARTS, SPADES, DIAMONDS }
```

Since enum classes *are classes*, all other rules for formatting classes apply.

4.8.2   Variable declarations

4.8.2.1   One variable per declaration

Every variable declaration (field or local) declares only one variable: declarations such as `int a, b;` are not used.

4.8.2.2   Declared when needed, initialized as soon as possible

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.

4.8.3   Arrays

4.8.3.1   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}
```

4.8.3.2   No C-style array declarations

The square brackets form a part of the *type*, not the variable: String`[] args`, not `String args[]`.

4.8.4   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.

4.8.4.1.1   Indentation

As with any other block, the contents of a switch block are indented +4.

After a switch label, a newline appears, and the indentation level is increased +4, 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.

4.8.4.1.2   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);
}
```

4.8.4.1.3   The default case is present

Each switch statement includes a `default` statement group, even if it contains no code.

4.9 Annotations

Annotations applying to a class, method or constructor appear immediately after the documentation block, and each annotation is listed on a line of its own (that is, one annotation per line). These line breaks do not constitute line-wrapping (section on Line-wrapping), so the indentation level is not increased. Example:

```
@Override
@Nullable
public String getNameIfPresent() { ... }
```

There are no specific rules for formatting parameter and local variable annotations.

4.10 Comments

4.10.1   Block comment style

Block comments are indented at the same level as the surrounding code. They may be in `/* ... */` style or `// ...` style. However the `/* ... */` should be preferred. 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.

4.11 Modifiers

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

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

4.12 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`.

5   Naming

5.1 Rules common to all identifiers

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

In this style special prefixes or suffixes, like those seen in the examples `name_`, `mName`, `s_name` and `kName`, are not used.

5.2 Rules by identifier type

5.2.1   Package names

Package names are all lowercase, with consecutive words simply concatenated together (no underscores). For example, `com.example.deepspace`, not `com.example.deepSpace` or `com.example.deep_space`.

### 5.2.2	Class names

Class names are written in UpperCamelCase

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).

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

Test classes are named starting with the name of the class they are testing, and ending with Test. For example, HashTest or HashIntegrationTest. If they cannot be associated with only one class the class should be named according to its test purpose ending with Test.

### 5.2.3	Method names

Method names are written in lowerCamelCase.

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

### 5.2.4	Constant names

Constant names use `CONSTANT_CASE`: all uppercase letters, with words separated by underscores. But what *is* a constant, exactly?

Every constant is a static final field, but not all static final fields are constants. Before choosing constant case, consider whether the field really *feels like* a constant. For example, if any of that instance's observable state can change, it is almost certainly not a constant. Merely *intending* to never mutate the object is generally not enough. Examples:

```
// Constants
static final int NUMBER = 5;
static final ImmutableList<String> NAMES = ImmutableList.of("Ed", "Ann");
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 Logger logger = Logger.getLogger(MyClass.getName());
static final String[] nonEmptyArray = {"these", "can", "change"};
```

These names are typically nouns or noun phrases.

### 5.2.5	Non-constant field names

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

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

### 5.2.6	Parameter names

Parameter names are written in lowerCamelCase

One-character parameter names should be avoided.

5.2.7   Local variable names

Local variable names are written in lowerCamelCase, and can be abbreviated more liberally than other types of names.

However, one-character names should be avoided, except for temporary and looping variables.

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

5.2.8   Type variable names

Each type variable is named in one of two styles:

- A single capital letter, optionally followed by a single numeral (such as `E`, `T`, `X`, `T2`)
- A name in the form used for classes (see section Class names), followed by the capital letter `T` (examples: RequestT, FooBarT).

5.3 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, Google Style specifies the following (nearly) deterministic scheme.

Beginning with the prose form of the name:

1. Convert the phrase to plain ASCII and remove any apostrophes. For example, "Müller's algorithm" might become "Muellers algorithm".

2. Divide this result into words, splitting on spaces and any remaining punctuation (typically hyphens).

    o *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.

3. Now lowercase *everything* (including acronyms), then uppercase only the first character of:

    o ... each word, to yield *upper camel case*, or

    o ... each word except the first, to yield *lower camel case*

4. 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.

6 Programming Practices

6.1 @Override: always used

A method is marked with the `@Override` annotation whenever it is legal. This includes a class method overriding a superclass method, a class method implementing an interface method, and an interface method respecifying a superinterface method.

6.2 Caught exceptions: not ignored

Except as noted below, it is very rarely correct to do nothing in response to a caught exception. (Typical responses are to log it, or if it is considered "impossible", rethrow it as an AssertionError.)

When it truly is appropriate to take no action whatsoever in a catch block, the reason this is justified is explained in a comment.

```
try {
  int i = Integer.parseInt(response);
  return handleNumericResponse(i);
} catch (NumberFormatException ok) {
  // it's not numeric; that's fine, just continue
}
return handleTextResponse(response);
```

Exception: In tests, a caught exception may be ignored without comment *if* it is named `expected`. The following is a very common idiom for ensuring that the method under test *does* throw an exception of the expected type, so a comment is unnecessary here.

```
try {
  emptyStack.pop();
  fail();
} catch (NoSuchElementException expected) {
}
```

6.3 Static members: qualified using class

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
```

6.4 Finalizers: not used

It is extremely rare to override Object.`finalize`.

Tip: Don't do it. If you absolutely must, be sure you understand what to do.

7 Javadoc

7.1 Formatting

7.1.1   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 there are no at-clauses present, and the entirety of the Javadoc block (including comment markers) can fit on a single line.

7.1.2   Paragraphs

One blank line—that is, a line containing only the aligned leading asterisk (`*`)—appears between paragraphs, and before the group of "at-clauses" if present.

7.1.3   At-clauses

Any of the standard "at-clauses" that are used appear in the order `@param`, `@return`, `@throws`, `@deprecated`, and these four types never appear with an empty description.

7.2 The summary fragment

The Javadoc for each class and member 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 may 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. */`.

7.3 Where Javadoc is used

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

Other classes and members still have Javadoc *as needed*. Whenever an implementation comment would be used to define the overall purpose or behavior of a class, method or field, that comment is written as Javadoc instead. (It's more uniform, and more tool-friendly.)

7.3.1   Exception: overrides

Javadoc is not always present on a method that overrides a supertype method.