

International School

**Capstone Project 1**

CMU-CS450

**Code Standard Document**

**Version 1.0**

**Date:** 30/08/2022

**Craft Village Pollution Monitor System**

**Submitted by**

**Ca, Van Cong Le**

**Huy, Bui Duc**

**Phuc, Hua Hoang**

**Trung, Nguyen Thanh**

**Approved by**

**Ph.D. Nguyen Thanh Binh**

**Proposal Review Panel Representative:**

Name Signature Date

**Capstone Project 1 - Mentor:**

Name Signature Date

**PROJECT INFORMATION**

|  |  |  |  |
| --- | --- | --- | --- |
| **Project acronym** | CVPMS | | |
| **Project Title** | Craft Village Pollution Monitor System | | |
| **Start Date** | 22/08/2022 | **End Date** | 07/12/2022 |
| **Lead Institution** | International School, Duy Tan University | | |
| **Project Mentor** | Ph.D. Thanh Binh, Nguyen | | |
| **Scrum master / Project Leader & contact details** | Ca, Van Cong Le  *Email:* [cascabusiness@gmail.com](about:blank)  *Tel:* 0352707895 | | |
| **Partner Organization** |  | | |
| **Project Web URL** | [*https://github.com/Casca113s2/craft-village-pollution-monitor-system*](https://github.com/Casca113s2/craft-village-pollution-monitor-system) | | |
| **Team members** | **Name** | **Email** | **Tel** |
| 25211207666 | Ca, Van Cong Le | [cascabusiness@gmail.com](http://cascabusiness@gmail.com) | 0352707895 |
| 25211215894 | Huy, Bui Duc | [duchuyltt122@gmail.com](http://duchuyltt122@gmail.com) | 0818648090 |
| 25211204084 | Phuc, Hua Hoang | [phuchuho0402@gmail.com](http://phuchuho0402@gmail.com) | 0905639682 |
| 25211215133 | Trung, Nguyen Thanh | [nguyenttrung2601@gmail.com](http://nguyenttrung2601@gmail.com) | 0774496838 |

DOCUMENT NAME

|  |  |  |  |
| --- | --- | --- | --- |
| **Document Title** | Code Standard Document | | |
| **Author(s)** | Van Cong Le Ca  Hua Hoang Phuc | | |
| **Date** | 18/09/2022 | **File Name** | C1SE.06\_CVPMS\_Code-Standard-Document\_v1.0.docx |

REVISION HISTORY

| **Version** | **Date** | **Comments** | **Author** | **Approval** |
| --- | --- | --- | --- | --- |
| 1.0 | 18/09/2022 | Initial Release | All members |  |

**Approve Document:** Sign in to approve the document

|  |  |  |  |
| --- | --- | --- | --- |
| **Mentor** | Binh, Nguyen Thanh | Date |  |
| Sign |  |
| **Scrum Master** | Ca, Van Cong Le | Date |  |
| Sign |  |
| **Scrum Member** | Huy, Bui Duc | Date |  |
| Sign |  |
| **Scrum Member** | Phuc, Hua Hoang | Date |  |
| Sign |  |
| **Scrum Member** | Trung, Nguyen Thanh | Date |  |
| Sign |  |

Table Of Contents

[1. Introduction 6](#_Toc121936623)

[1.1. Purpose 6](#_Toc121936624)

[1.2. Scope 6](#_Toc121936625)

[2. Code Standards 6](#_Toc121936626)

[2.1. Dart Language Code Standard 6](#_Toc121936627)

[2.1.1. Identifiers 6](#_Toc121936628)

[2.1.2. Ordering 6](#_Toc121936629)

[2.1.3. Formatting 6](#_Toc121936630)

[2.1.4. Comments 7](#_Toc121936631)

[2.1.5. Doc comments 7](#_Toc121936632)

[2.1.6. Markdown 7](#_Toc121936633)

[2.1.7. Writing 7](#_Toc121936634)

[2.1.8. Libraries 8](#_Toc121936635)

[2.1.9. Null 8](#_Toc121936636)

[2.1.10. Strings 8](#_Toc121936637)

[2.1.11. Functions 8](#_Toc121936638)

[2.1.12. Members 9](#_Toc121936639)

[2.1.13. Constructors 9](#_Toc121936640)

[2.1.14. Error handling 9](#_Toc121936641)

[2.1.15. Asynchrony 9](#_Toc121936642)

[2.1.16. Names 9](#_Toc121936643)

[2.1.17. Libraries 10](#_Toc121936644)

[2.1.18. Classes and mixins 10](#_Toc121936645)

[2.1.19. Constructors 10](#_Toc121936646)

[2.1.20. Types 11](#_Toc121936647)

[2.1.21. Parameters 11](#_Toc121936648)

[2.2. Java Language Code Standard 12](#_Toc121936649)

[2.2.1. Source file structure 12](#_Toc121936650)

[2.2.2. Formatting 12](#_Toc121936651)

[2.2.3. Naming 14](#_Toc121936652)

[2.2.4. Programming Practices 15](#_Toc121936653)

[3. References 15](#_Toc121936654)

# Introduction

## Purpose

This Coding Standard requires certain practices for developing programs in the Java, Dart language. The objective of this coding standard is to have a positive effect on:

* Avoidance of errors/bugs, especially the hard-to-find ones.
* Maintainability, by promoting some proven design principles

## Scope

This standard pertains to the use of the Java, Dart language.

# Code Standards

## Dart Language Code Standard

### Identifiers

* DO name types using UpperCamelCase.
* DO name extensions using UpperCamelCase.
* DO name libraries, packages, directories, and source files using lowercase\_with\_underscores.
* DO name import prefixes using lowercase\_with\_underscores.
* DO name other identifiers using lowerCamelCase.
* PREFER using lowerCamelCase for constant names.
* DO capitalize acronyms and abbreviations longer than two letters like words.
* PREFER using \_, \_\_, etc. for unused callback parameters.
* DON’T use a leading underscore for identifiers that aren’t private.
* DON’T use prefix letters.

### Ordering

* DO place “dart:” imports before other imports.
* DO place “package:” imports before relative imports.
* DO specify exports in a separate section after all imports.
* DO sort sections alphabetically.

### Formatting

* DO format your code using dart format.
* CONSIDER changing your code to make it more formatter-friendly.
* AVOID lines longer than 80 characters.
* DO use curly braces for all flow control statements.

### Comments

* DO format comments like sentences.
* DON’T use block comments for documentation.

### Doc comments

* DO use /// doc comments to document members and types.
* PREFER writing doc comments for public APIs.
* CONSIDER writing a library-level doc comment.
* CONSIDER writing doc comments for private APIs.
* DO start doc comments with a single-sentence summary.
* DO separate the first sentence of a doc comment into its own paragraph.
* AVOID redundancy with the surrounding context.
* PREFER starting function or method comments with third-person verbs.
* PREFER starting a non-boolean variable or property comment with a noun phrase.
* PREFER starting a boolean variable or property comment with “Whether” followed by a noun or gerund phrase.
* DON’T write documentation for both the getter and setter of a property.
* PREFER starting library or type comments with noun phrases.
* CONSIDER including code samples in doc comments.
* DO use square brackets in doc comments to refer to in-scope identifiers.
* DO use prose to explain parameters, return values, and exceptions.
* DO put doc comments before metadata annotations.

### Markdown

* AVOID using markdown excessively.
* AVOID using HTML for formatting.
* PREFER backtick fences for code blocks.

### Writing

* PREFER brevity.
* AVOID abbreviations and acronyms unless they are obvious.
* PREFER using “this” instead of “the” to refer to a member’s instance.

### Libraries

* DO use strings in part of directives.
* DON’T import libraries that are inside the src directory of another package.
* DON’T allow an import path to reach into or out of lib.
* PREFER relative import paths.

### Null

* DON’T explicitly initialize variables to null.
* DON’T use an explicit default value of null.
* PREFER using ?? to convert null to a boolean value.
* AVOID late variables if you need to check whether they are initialized.
* CONSIDER assigning a nullable field to a local variable to enable type promotion.

### Strings

* DO use adjacent strings to concatenate string literals.
* PREFER using interpolation to compose strings and values.
* AVOID using curly braces in interpolation when not needed.
* Collections
* DO use collection literals when possible.
* DON’T use .length to see if a collection is empty.
* AVOID using Iterable.forEach() with a function literal.
* DON’T use List.from() unless you intend to change the type of the result.
* DO use whereType() to filter a collection by type.
* DON’T use cast() when a nearby operation will do.
* AVOID using cast().

### Functions

* DO use a function declaration to bind a function to a name.
* DON’T create a lambda when a tear-off will do.
* DO use = to separate a named parameter from its default value.
* Variables
* DO follow a consistent rule for var and final on local variables.
* AVOID storing what you can calculate.

### Members

* DON’T wrap a field in a getter and setter unnecessarily.
* PREFER using a final field to make a read-only property.
* CONSIDER using => for simple members.
* DON’T use this. except to redirect to a named constructor or to avoid shadowing.
* DO initialize fields at their declaration when possible.

### Constructors

* DO use initializing formals when possible.
* DON’T use late when a constructor initializer list will do.
* DO use ; instead of {} for empty constructor bodies.
* DON’T use new.
* DON’T use const redundantly.

### Error handling

* AVOID catches without on clauses.
* DON’T discard errors from catches without on clauses.
* DO throw objects that implement Error only for programmatic errors.
* DON’T explicitly catch Error or types that implement it.
* DO use rethrow to rethrow a caught exception.

### Asynchrony

* PREFER async/await over using raw futures.
* DON’T use async when it has no useful effect.
* CONSIDER using higher-order methods to transform a stream.
* AVOID using Completer directly.
* DO test for Future<T> when disambiguating a FutureOr<T> whose type argument could be Object.

### Names

* DO use terms consistently.
* AVOID abbreviations.
* PREFER putting the most descriptive noun last.
* CONSIDER making the code read like a sentence.
* PREFER a noun phrase for a non-boolean property or variable.
* PREFER a non-imperative verb phrase for a boolean property or variable.
* CONSIDER omitting the verb for a named boolean parameter.
* PREFER the “positive” name for a boolean property or variable.
* PREFER an imperative verb phrase for a function or method whose main purpose is a side effect.
* PREFER a noun phrase or non-imperative verb phrase for a function or method if returning a value is its primary purpose.
* CONSIDER an imperative verb phrase for a function or method if you want to draw attention to the work it performs.
* AVOID starting a method name with get.
* PREFER naming a method to\_\_\_() if it copies the object’s state to a new object.
* PREFER naming a method as\_\_\_() if it returns a different representation backed by the original object.
* AVOID describing the parameters in the function’s or method’s name.
* DO follow existing mnemonic conventions when naming type parameters.

### Libraries

* PREFER making declarations private.
* CONSIDER declaring multiple classes in the same library.

### Classes and mixins

* AVOID defining a one-member abstract class when a simple function will do.
* AVOID defining a class that contains only static members.
* AVOID extending a class that isn’t intended to be subclassed.
* DO document if your class supports being extended.
* AVOID implementing a class that isn’t intended to be an interface.
* DO document if your class supports being used as an interface.
* DO use mixin to define a mixin type.
* AVOID mixing in a type that isn’t intended to be a mixin.

### Constructors

* CONSIDER making your constructor const if the class supports it.
* Members
* PREFER making fields and top-level variables final.
* DO use getters for operations that conceptually access properties.
* DO use setters for operations that conceptually change properties.
* DON’T define a setter without a corresponding getter.
* AVOID using runtime type tests to fake overloading.
* AVOID public late final fields without initializers.
* AVOID returning nullable Future, Stream, and collection types.
* AVOID returning this from methods just to enable a fluent interface.

### Types

* DO type annotate variables without initializers.
* DO type annotate fields and top-level variables if the type isn’t obvious.
* DON’T redundantly type annotate initialized local variables.
* DO annotate return types on function declarations.
* DO annotate parameter types on function declarations.
* DON’T annotate inferred parameter types on function expressions.
* DON’T type annotate initializing formals.
* DO write type arguments on generic invocations that aren’t inferred.
* DON’T write type arguments on generic invocations that are inferred.
* AVOID writing incomplete generic types.
* DO annotate with dynamic instead of letting inference fail.
* PREFER signatures in function type annotations.
* DON’T specify a return type for a setter.
* DON’T use the legacy typedef syntax.
* PREFER inline function types over typedefs.
* PREFER using function type syntax for parameters.
* AVOID using dynamic unless you want to disable static checking.
* DO use Future<void> as the return type of asynchronous members that do not produce values.
* AVOID using FutureOr<T> as a return type.

### Parameters

* AVOID positional boolean parameters.
* AVOID optional positional parameters if the user may want to omit earlier parameters.
* AVOID mandatory parameters that accept a special “no argument” value.
* DO use inclusive start and exclusive end parameters to accept a range.
* Equality
* DO override hashCode if you override ==.
* DO make your == operator obey the mathematical rules of equality.
* AVOID defining custom equality for mutable classes.
* DON’T make the parameter to == nullable.

## Java Language Code Standard

### Source file structure

* License or copyright information, if present: If license or copyright information belongs in a file, it belongs here.
* Package statement: The package statement is not line-wrapped. The column limit does not apply to package statements.
* Import statements
  + No wildcard imports
  + No line-wrapping
  + Ordering and spacing
  + No static import for classes
* Class declaration
  + Exactly one top-level class declaration
  + Ordering of class contents

### Formatting

* Braces
  + Use of optional braces
  + Nonempty blocks: K & R style
  + Empty blocks: may be concise
* Block indentation: +2 spaces
* One statement per line
* Column limit: 100
* Line-wrapping
  + Prefer to break at a higher syntactic level
  + Indent continuation lines at least +4 spaces
* Whitespace
  + Vertical Whitespace
  + Horizontal whitespace
  + Horizontal alignment: never required
* Grouping parentheses: recommended
* Specific constructs
  + Enum classes
  + Variable declarations
    - One variable per declaration
    - Declared when needed
  + Arrays
    - Array initializers: can be "block-like"
    - No C-style array declarations
  + Switch statements
    - Indentation
    - Fall-through: commented
    - Presence of the *default* label
  + Annotations
    - Type-use annotations
    - Class annotations
    - Method and constructor annotations
    - Field annotations
    - Parameter and local variable annotations
  + Comments
    - Block comment style: Block comments are indented at the same level as the surrounding code.
  + Modifiers: Class and member modifiers, when present, appear in the order recommended by the Java Language Specification
  + Numeric Literals: long-valued integer literals use an uppercase L suffix, never lowercase (to avoid confusion with the digit 1).

### Naming

* Rules common to all identifiers: Identifiers use only ASCII letters and digits, and, in a small number of cases noted below, underscores.
* Rules by identifier type
  + Package names use only lowercase letters and digits (no underscores). Consecutive words are simply concatenated together.
  + Class names are written in UpperCamelCase. Class names are typically nouns or noun phrases
  + Method names are written in lowerCamelCase. Method names are typically verbs or verb phrases.
  + Constant names use UPPER\_SNAKE\_CASE: all uppercase letters, with each word separated from the next by a single underscore.
  + Non-constant field names (static or otherwise) are written in lowerCamelCase. These names are typically nouns or noun phrases.
  + Parameter names are written in lowerCamelCase. One-character parameter names in public methods should be avoided.
  + Local variable names are written in lowerCamelCase. Even when final and immutable, local variables are not considered to be constants, and should not be styled as constants.
  + 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 5.2.2, Class names), followed by the capital letter T (examples: RequestT, FooBarT).
* Camel case: defined
  + 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.

### Programming Practices

* @Override: always used
* Caught exceptions: not ignored
* Static members: qualified using class
* Finalizers: not used

# References

1. *Google team*, [“Effective Dart”](https://dart.dev/guides/language/effective-dart/style)
2. *Google team*, [“Google Java Style Guide”](https://google.github.io/styleguide/javaguide.html)