Flutter (Dart) Coding Standards Documentation

Version: 1.0.0

Status: Pending

Date: 2024/04/23

Prepared by:			
Name	**Metric number**	**Department**	**Date**
Koo Song Le	S2116593	QA (Standard & Compliance)	
Zhang Wei	S2004131	QA (Standard & Compliance)	
Approved by:			
Name	**Metric number**	**Department**	**Date**
Released by:			
Name	**Metric number**	**Department**	**Date**

Version History

Version	Release Date	Section	Amendments
1.0.0		All	Original Document

Change Record

Version	Date	Affected Section	Change Request #	Reason/Initiation/Remarks

Table of Contents

1.0 Description

1.1 Purpose

This document, primarily referenced to established Dart documentation and tools, specifies standards and best practices for coding and documenting Flutter (Dart) code for the development of the Optical Character Recognition (OCR) System for Handwritten Multiple-Choice Question (MCQ) Answer Sheet Evaluation project.

This document outlines the coding standards, design principles, and naming conventions specifically for Flutter (Dart) development projects. It serves as a reference for the software development teams involved in this project, facilitating the creation of efficient, maintainable and consistent code.

1.2 Scope

All Flutter (Dart) code checked into the OCR software-development project' permanent repository shall adhere to the standards and best practices specified in this document and its references.

1.3 Usage

All Flutter (Dart) code should be run through the official Dart formatter (`dart format`).

2.0 Related Documents

2.1 References

Dart (2023). *Effective Dart: Design.* https://dart.dev/effective-dart/design

2.2 Abbreviations and Acronyms

Abbreviation/Acronym	Description

2.3 Glossary

3.0 Code Standards

3.1 Naming Convention

3.1.1 Descriptive Names

The names assigned shall be meaningful and accurately describes the purpose or role of a variable, function, class, or module.

```
good
  pageCount         // A field.
  updatePageCount() // Consistent with pageCount.
  toSomething()     // Consistent with Iterable's toList().
  asSomething()     // Consistent with List's asMap().
  Point             // A familiar concept.
```

```
bad
  renumberPages()      // Confusingly different from pageCount.
  convertToSomething() // Inconsistent with toX() precedent.
  wrappedAsSomething() // Inconsistent with asX() precedent.
  Cartesian            // Unfamiliar to most users.
```

3.1.2 “lowerCamelCase” for all identifiers

All identifiers except for 3.1.3 and 3.1.4 shall follow the “lowerCamelCase” convention, where the term starts with a lowercase letter, every word after the first starts with uppercase letter, and/or no underscores between the words. This includes constant names.

```
good
  var count = 3;

  HttpRequest httpRequest;

  void align(bool clearItems) {
    // ...
  }
```

3.1.3 “snake_case” for packages, directories, source file name, and import prefixes

Packages, directories, and source file names, as well as import prefixes shall follow the snake_case convention, which is the words in lowercase form and/or separated by underscores (_).

good

```
my_package
└ lib
   └ file_system.dart
   └ slider_menu.dart
```

bad

```
mypackage
└ lib
   └ file-system.dart
   └ SliderMenu.dart
```

good

```dart
import 'dart:math' as math;
import 'package:angular_components/angular_components.dart' as angular_components;
import 'package:js/js.dart' as js;
```

bad

```dart
import 'dart:math' as Math;
import 'package:angular_components/angular_components.dart' as angularComponents;
import 'package:js/js.dart' as JS;
```

3.1.4 “CamelCase” for types and extensions names

Class names shall follow the “CamelCase” conventions, which the words start with an uppercase letter and/or with no underscores between the words.

good

```dart
class SliderMenu { ... }

class HttpRequest { ... }

typedef Predicate<T> = bool Function(T value);
```

good

```dart
extension MyFancyList<T> on List<T> { ... }

extension SmartIterable<T> on Iterable<T> { ... }
```

3.1.5 Reserved keywords

Dart reserved keywords shall not be used as identifiers.

3.2 Code Formatting

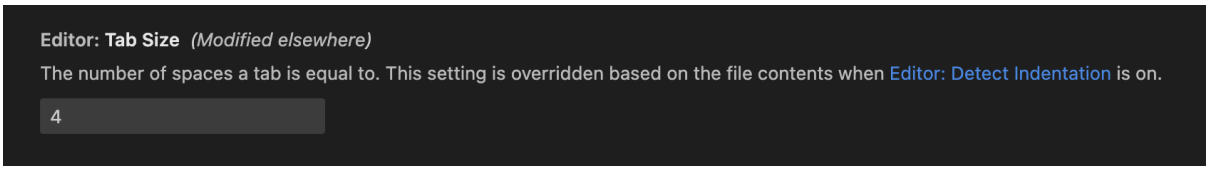
3.2.1 Indentation

Two spaces shall be used for each level of indentation.

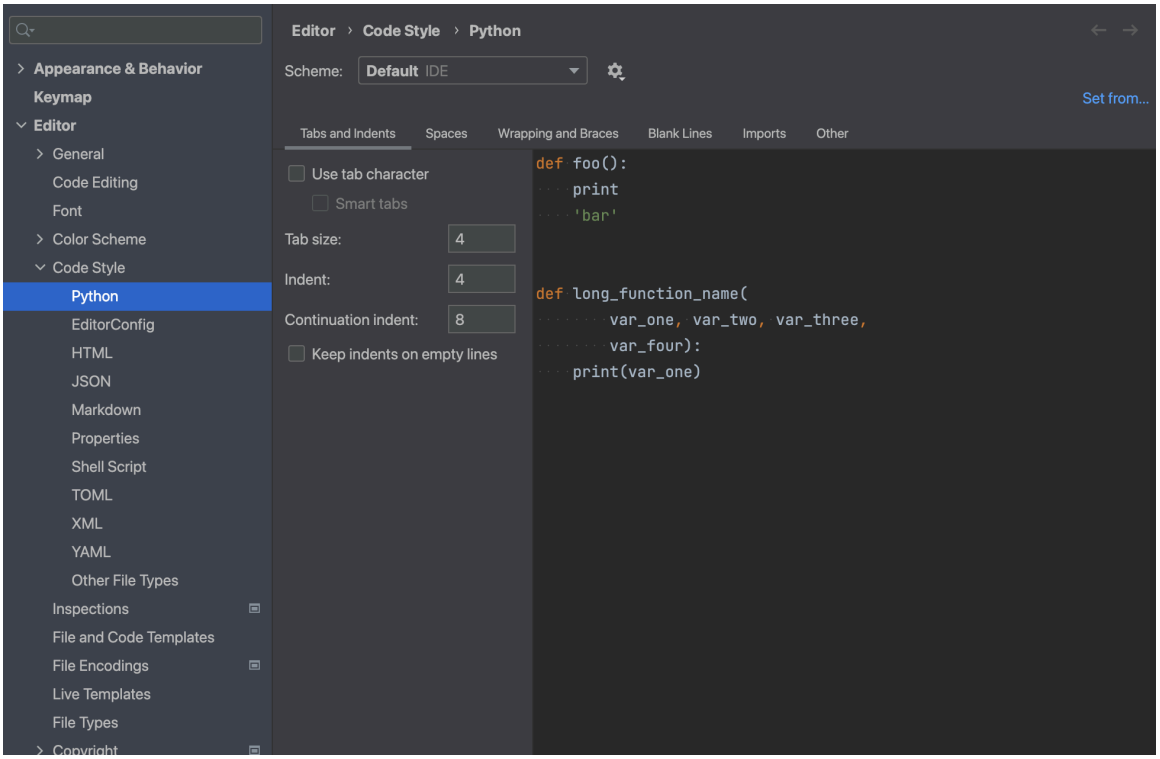
```
// good
main() {
  first(statement);
  second(statement);
}

// bad
main() {
    first(statement);
    second(statement);
}
```

To change tab size in Visual Studio Code, proceed to **Settings** > **Commonly Used** > **Editor: Tab Size**



To change tab size in PyCharm, proceed to **Settings** > **Editor** > **Code Style** > **Python** > **Tab size**



For braces/brackets/parenthesis alignment, the closing brace/bracket/parenthesis shall be aligned with the first character of the line that starts the construct.

```
class Foo {
  method() {
    if (someCondition) {
      // ...
    } else {
      // ...
    }
  }
}
```

3.2.2 Line Length

The lines of code shall be kept to a maximum of 79 characters per line of code. If the line of code is longer or exceeded the set number of characters, the code shall be broken into multiple lines, adhering to logical indentations.

```
# Good example
def long_function_name(
        parameter1, parameter2, parameter3,
        parameter4, parameter5, parameter6):
    pass


# Bad example
def long_function_name(parameter1, parameter2, parameter3, parameter4, parameter5, parameter6):
    pass
```

3.2.3 Blank Lines

Blank lines shall be used to separate logical sections of a code, such as functions, classes, or blocks of related code. Two blank lines shall be used between top-level definitions instead.

```
main() {
  first(statement);
  second(statement);

  third(statement);
}

anotherDeclaration() { ... }
```

3.2.4 Whitespaces

3.2.4.1 Avoid Extraneous Whitespace

Avoid unnecessary spaces in the following scenarios:

- Before/after values in parentheses, brackets, or braces

```
var numbers = <int>[1, 2, (3 + 4)];
```

- Around a unary operator

```
!condition
index++
```

- Between the declared name of a method, operator, or setter and its parameter list.

```
log(arg) { ... }
bool operator ==(other) => ...
set contents(value) { ... }
```

3.2.4.2 Add Whitespace

Add whitespaces for the following scenarios:

- After control flow keywords

```
while (foo) { ... }

try {
  // ...
} catch (e) {
  // ...
}
```

- Around binary and ternary operators

```
average = (a + b) / 2;
largest = a > b ? a : b;
if (obj is! SomeType) print('not SomeType');
```

- After the ‘operator’ keyword

```
bool operator ==(other) => ...
```

- After , and : when used in a map or named parameter

```
function(a, b, named: c)
[some, list, literal]
{map: literal}
```

- Around ‘in’ and after each ; in a loop

```
for (var i = 0; i < 100; i++) { ... }

for (final item in collection) { ... }
```

- Before { in function and method bodies.

```
getEmptyFn(a) {
  return () {};
}
```

3.2.4.3 Other Recommendations

- Remove trailing whitespace as it can be misleading and is not preserved in all environments.
- Use spaces around the equal sign (=) when combining argument annotations with default values

3.3 Documentation and Comments

3.3.1 Documentation comments (doc comments)

Use /// for doc comments.

```
good
  /// The number of characters in this chunk when unsplit.
  int get length => ...
```

```
bad
  // The number of characters in this chunk when unsplit.
  int get length => ...
```

If someone could have written the same documentation without knowing anything about the class other than its name, then it's useless.

Avoid checking in such documentation, because it is no better than no documentation but will prevent us from noticing that the identifier is not actually documented.

```
// BAD:

/// The background color.
final Color backgroundColor;

/// Half the diameter of the circle.
final double radius;


// GOOD:

/// The color with which to fill the circle.
///
/// Changing the background color will cause the avatar to animate to the new color.
final Color backgroundColor;

/// The size of the avatar.
///
/// Changing the radius will cause the avatar to animate to the new size.
final double radius;
```

3.3.2 Block Comments

Do not use block comments (/* ... */).

```
bad
  void greet(String name) {
    /* Assume we have a valid name. */
    print('Hi, $name!');
  }
```

3.3.3 Inline Comments

Use // for regular comments. Format comments like sentences.

```
good
  // Not if anything comes before it.
  if (_chunks.isNotEmpty) return false;
```

3.4 Imports, Packages, and Libraries

3.4.1 Imports

Imports should usually be on separate lines and put at the top of the file, just after any module comments and doc comments.

```
good
  import 'dart:async';
  import 'dart:html';

  import 'package:bar/bar.dart';
  import 'package:foo/foo.dart';

  import 'util.dart';

  export 'src/error.dart';
```

Imports should be grouped in the following order:

1. dart: imports
2. package: imports
3. other import utilities
4. exports

A blank line should be put between each group of imports.

Prefer relative imports, unless it reaches across `lib` or `src`.

lib/api.dart

```
good
  import 'src/stuff.dart';
  import 'src/utils.dart';
```

lib/src/utils.dart

```
good
  import '../api.dart';
  import 'stuff.dart';
```

test/api_test.dart

```
good
  import 'package:my_package/api.dart'; // Don't reach into 'lib'.

  import 'test_utils.dart'; // Relative within 'test' is fine.
```

3.4.2 Libraries

If using `part of` directive to use a library, use the URI string to the library file, not the library name:

```
good
  part of '../../my_library.dart';
```

```
bad
  part of my_library;
```

3.5 Others

3.5.1 Nulls

Do not explicitly initialise null.

good

```
Item? bestDeal(List<Item> cart) {
  Item? bestItem;

  for (final item in cart) {
    if (bestItem == null || item.price < bestItem.price) {
      bestItem = item;
    }
  }

  return bestItem;
}
```

bad

```
Item? bestDeal(List<Item> cart) {
  Item? bestItem = null;

  for (final item in cart) {
    if (bestItem == null || item.price < bestItem.price) {
      bestItem = item;
    }
  }

  return bestItem;
}
```

good

```
void error([String? message]) {
  stderr.write(message ?? '\n');
}
```

bad

```
void error([String? message = null]) {
  stderr.write(message ?? '\n');
}
```

3.5.2 Strings

Use adjacent strings to join string literals together without +.

```
good
  raiseAlarm('ERROR: Parts of the spaceship are on fire. Other '
      'parts are overrun by martians. Unclear which are which.');
```

```
bad
  raiseAlarm('ERROR: Parts of the spaceship are on fire. Other ' +
      'parts are overrun by martians. Unclear which are which.');
```

Prefer interpolation when combining multiple literals and values.

```
good
  'Hello, $name! You are ${year - birth} years old.';
```

```
bad
  'Hello, ' + name + '! You are ' + (year - birth).toString() + ' y...';
```

Avoid using curly braces in interpolation when not needed

```
good
  var greeting = 'Hi, $name! I love your ${decade}s costume.';
```

```
bad
  var greeting = 'Hi, ${name}! I love your ${decade}s costume.';
```

3.5.3 Unspecified formatting/code style

Use `dart format` as the definitive guideline for formatting Flutter/Dart code.