Effective Dart: Style

A surprisingly important part of good code is good style. Consistent naming, ordering, and formatting helps code that *is* the same *look* the same. It takes advantage of the powerful pattern-matching hardware most of us have in our ocular systems. If we use a consistent style across the entire Dart ecosystem, it makes it easier for all of us to learn from and contribute to each others' code.

Identifiers

Identifiers come in three flavors in Dart.

- `UpperCamelCase` names capitalize the first letter of each word, including the first.
- `lowerCamelCase` names capitalize the first letter of each word, *except* the first which is always lowercase, even if it's an acronym.
- `lowercase_with_underscores` names use only lowercase letters, even for acronyms, and separate words with `_`.

DO name types using `UpperCamelCase`.

Linter rule: camel_case_types

Classes, enum types, typedefs, and type parameters should capitalize the first letter of each word (including the first word), and use no separators.

good

```
class SliderMenu { ... }

class HttpRequest { ... }

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

This even includes classes intended to be used in metadata annotations.

good

```
class Foo {
  const Foo([arg]);
}

@Foo(anArg)
class A { ... }

@Foo()
class B { ... }
```

If the annotation class's constructor takes no parameters, you might want to create a separate `lowerCamelCase` constant for it.

good

```
const foo = Foo();

@foo
class C { ... }
```

DO name extensions using `UpperCamelCase`.

Linter rule: camel_case_extensions

Like types, extensions should capitalize the first letter of each word (including the first word), and use no separators.

good

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

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

> **Version note:** Extensions are a Dart 2.7 language feature. For details, see the extensions design document.

DO name libraries, packages, directories, and source files using `lowercase_with_underscores`.

Linter rules: library_names, file_names

Some file systems are not case-sensitive, so many projects require filenames to be all lowercase. Using a separating character allows names to still be readable in that form. Using underscores as the separator ensures that the name is still a valid Dart identifier, which may be helpful if the language later supports symbolic imports.

good

```
library peg_parser.source_scanner;

import 'file_system.dart';
import 'slider_menu.dart';
```

bad

```
library pegparser.SourceScanner;

import 'file-system.dart';
import 'SliderMenu.dart';
```

> **Note:** This guideline specifies *how* to name a library *if you choose to name it*. It is fine to *omit* the library directive in a file if you want.

DO name import prefixes using `lowercase_with_underscores`.

Linter rule: library_prefixes

good

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

bad

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

DO name other identifiers using `lowerCamelCase`.

Linter rule: non_constant_identifier_names

Class members, top-level definitions, variables, parameters, and named parameters should capitalize the first letter of each word *except* the first word, and use no separators.

good

```
var count = 3;

HttpRequest httpRequest;

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

PREFER using `lowerCamelCase` for constant names.

Linter rule: constant_identifier_names

In new code, use `lowerCamelCase` for constant variables, including enum values.

good

```
const pi = 3.14;
const defaultTimeout = 1000;
final urlScheme = RegExp('^([a-z]+):');

class Dice {
  static final numberGenerator = Random();
}
```

bad

```
const PI = 3.14;
const DefaultTimeout = 1000;
final URL_SCHEME = RegExp('^([a-z]+):');

class Dice {
  static final NUMBER_GENERATOR = Random();
}
```

You may use `SCREAMING_CAPS` for consistency with existing code, as in the following cases:

- When adding code to a file or library that already uses `SCREAMING_CAPS`.
- When generating Dart code that's parallel to Java code — for example, in enumerated types generated from protobufs.

> **Note:** We initially used Java's `SCREAMING_CAPS` style for constants. We changed for a few reasons:
>
> - `SCREAMING_CAPS` looks bad for many cases, particularly enum values for things like CSS colors.
> - Constants are often changed to final non-const variables, which would necessitate a name change.
> - The `values` property automatically defined on an enum type is const and lowercase.

DO capitalize acronyms and abbreviations longer than two letters like words.

Capitalized acronyms can be hard to read, and multiple adjacent acronyms can lead to ambiguous names. For example, given a name that starts with `HTTPSFTP`, there's no way to tell if it's referring to HTTPS FTP or HTTP SFTP.

To avoid this, acronyms and abbreviations are capitalized like regular words.

Exception: Two-letter *acronyms* like IO (input/output) are fully capitalized: `IO`. On the other hand, two-letter *abbreviations* like ID (identification) are still capitalized like regular words: `Id`.

good

```
class HttpConnection {}
class DBIOPort {}
class TVVcr {}
class MrRogers {}

var httpRequest = ...
var uiHandler = ...
Id id;
```

bad

```
class HTTPConnection {}
class DbIoPort {}
class TvVcr {}
class MRRogers {}

var hTTPRequest = ...
var uIHandler = ...
ID iD;
```

PREFER using _, __, etc. for unused callback parameters.

Sometimes the type signature of a callback function requires a parameter, but the callback implementation doesn't *use* the parameter. In this case, it's idiomatic to name the unused parameter `_`. If the function has multiple unused parameters, use additional underscores to avoid name collisions: `__`, `___`, etc.

good

```
futureOfVoid.then((_) {
  print('Operation complete.');
});
```

This guideline is only for functions that are both *anonymous and local*. These functions are usually used immediately in a context where it's clear what the unused parameter represents. In contrast, top-level functions and method declarations don't have that context, so their parameters must be named so that it's clear what each parameter is for, even if it isn't used.

DON'T use a leading underscore for identifiers that aren't private.

Dart uses a leading underscore in an identifier to mark members and top-level declarations as private. This trains users to associate a leading underscore with one of those kinds of declarations. They see "_" and think "private".

There is no concept of "private" for local variables, parameters, local functions, or library prefixes. When one of those has a name that starts with an underscore, it sends a confusing signal to the reader. To avoid that, don't use leading underscores in those names.

DON'T use prefix letters.

Hungarian notation and other schemes arose in the time of BCPL, when the compiler didn't do much to help you understand your code. Because Dart can tell you the type, scope, mutability, and other properties of your declarations, there's no reason to encode those properties in identifier names.

good

```
defaultTimeout
```

bad

```
kDefaultTimeout
```

Ordering

To keep the preamble of your file tidy, we have a prescribed order that directives should appear in. Each "section" should be separated by a blank line.

A single linter rule handles all the ordering guidelines: directives_ordering.

DO place “dart:” imports before other imports.

Linter rule: directives_ordering

good

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

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

DO place “package:” imports before relative imports.

Linter rule: directives_ordering

good

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

import 'util.dart';
```

DO specify exports in a separate section after all imports.

Linter rule: directives_ordering

good

```
import 'src/error.dart';
import 'src/foo_bar.dart';

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

bad

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

DO sort sections alphabetically.

Linter rule: directives_ordering

good

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

import 'foo.dart';
import 'foo/foo.dart';
```

bad

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

import 'foo/foo.dart';
import 'foo.dart';
```

Formatting

Like many languages, Dart ignores whitespace. However, *humans* don’t. Having a consistent whitespace style helps ensure that human readers see code the same way the compiler does.

DO format your code using `dart format`.

Formatting is tedious work and is particularly time-consuming during refactoring. Fortunately, you don't have to worry about it. We provide a sophisticated automated code formatter called `dart format` that does it for you. We have some documentation on the rules it applies, but the official whitespace-handling rules for Dart are *whatever `dart format` produces*.

The remaining formatting guidelines are for the few things `dart format` cannot fix for you.

CONSIDER changing your code to make it more formatter-friendly.

The formatter does the best it can with whatever code you throw at it, but it can't work miracles. If your code has particularly long identifiers, deeply nested expressions, a mixture of different kinds of operators, etc. the formatted output may still be hard to read.

When that happens, reorganize or simplify your code. Consider shortening a local variable name or hoisting out an expression into a new local variable. In other words, make the same kinds of modifications that you'd make if you were formatting the code by hand and trying to make it more readable. Think of `dart format` as a partnership where you work together, sometimes iteratively, to produce beautiful code.

AVOID lines longer than 80 characters.

Linter rule: lines_longer_than_80_chars

Readability studies show that long lines of text are harder to read because your eye has to travel farther when moving to the beginning of the next line. This is why newspapers and magazines use multiple columns of text.

If you really find yourself wanting lines longer than 80 characters, our experience is that your code is likely too verbose and could be a little more compact. The main offender is usually `VeryLongCamelCaseClassNames`. Ask yourself, "Does each word in that type name tell me something critical or prevent a name collision?" If not, consider omitting it.

Note that `dart format` does 99% of this for you, but the last 1% is you. It does not split long string literals to fit in 80 columns, so you have to do that manually.

Exception: When a URI or file path occurs in a comment or string (usually in an import or export), it may remain whole even if it causes the line to go over 80 characters. This makes it easier to search source files for a path.

Exception: Multi-line strings can contain lines longer than 80 characters because newlines are significant inside the string and splitting the lines into shorter ones can alter the program.

DO use curly braces for all flow control statements.

Linter rule: curly_braces_in_flow_control_structures

Doing so avoids the dangling else problem.

good

```
if (isWeekDay) {
  print('Bike to work!');
} else {
  print('Go dancing or read a book!');
}
```

Exception: When you have an `if` statement with no `else` clause and the whole `if` statement fits on one line, you can omit the braces if you prefer:

good

```
if (arg == null) return defaultValue;
```

If the body wraps to the next line, though, use braces:

good

```
if (overflowChars != other.overflowChars) {
  return overflowChars < other.overflowChars;
}
```

bad

```
if (overflowChars != other.overflowChars)
  return overflowChars < other.overflowChars;
```