Código de buenas prácticas

Dart - Flutter

INTEGRANTES GRUPO 7:

Apellido y Nombre	Legajo	Mail
Salinas Luciano Exequiel	73212	exesalinas393@gmail.com
Brizuela Marcelo Ismael	53756	marcelo.brizuela468@gmail.com
Medina Juan Cruz	80405	Juancruzfaq@gmail.com
Perez Leonardo Federico	81295	fenixi1234567890@gmail.com

Una parte importante de un buen código es el buen estilo de escritura: Nombrar, ordenar y formatear consistentemente ayuda a que el código que *es* igual se *vea* igual. Aprovecha el poderoso hardware de coincidencia de patrones que la mayoría de nosotros tenemos en nuestros sistemas oculares. Si usamos un estilo coherente en todo el ecosistema de Dart/Flutter, será más fácil para todos aprender y contribuir al código de los demás.

Identificadores

Los identificadores vienen en tres versiones en Dart.

- UpperCamelCase Los nombres escriben en mayúscula la primera letra de cada palabra, incluida la primera.
- lowerCamelCase Los nombres escriben en mayúscula la primera letra de cada palabra, *excepto* la primera que siempre está en minúsculas, incluso si es un acrónimo.
- lowercase_with_underscores los nombres usan solo letras minúsculas, incluso para acrónimos, y separan palabras con _.

Hacer los tipos de nombre que utilizan UpperCamelCase.

Las clases, los tipos de enumeración, las definiciones de tipo y los parámetros de tipo deben escribir en mayúscula la primera letra de cada palabra (incluida la primera palabra) y no deben utilizar separadores.

```
class SliderMenu { ... }

class HttpRequest { ... }

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

Esto incluye clases destinadas a ser utilizadas en anotaciones de metadatos.

```
class Foo {

  const Foo([Object? arg]);

}



@Foo(anArg)

class A { ... }



@Foo()
```

```
class B { ... }
```

Si el constructor de la clase de anotación no toma parámetros, es posible que desee crear una constante lowerCamelCase separada para él.

```
const foo = Foo();


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

DEBE utilizar extensiones de nombre UpperCamelCase.

Al igual que los tipos, las extensiones deben escribir en mayúscula la primera letra de cada palabra (incluida la primera palabra) y no deben usar separadores.

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

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

NOMBRE las bibliotecas, paquetes, directorios y archivos fuente usando lowercase_with_underscores.

Algunos sistemas de archivos no distinguen entre mayúsculas y minúsculas, por lo que muchos proyectos requieren que los nombres de los archivos estén en minúsculas. El uso de un carácter de separación permite que los nombres sigan siendo legibles en esa forma. El uso de guiones bajos como separador garantiza que el nombre siga siendo un identificador de Dart válido, lo que puede ser útil si el idioma admite importaciones simbólicas posteriormente.

```
library peg_parser.source_scanner;


import 'file_system.dart';
```

```
import 'slider_menu.dart';

library pegparser.SourceScanner;

import 'file-system.dart';

import 'SliderMenu.dart';
```

Nota: Esta guía especifica *cómo* nombrar una biblioteca *si elige nombrarla* . Está bien *omitir* la directiva de biblioteca en un archivo si lo desea.

DEBE nombrar prefijos de importación usando lowercase_with_underscores.

```
import 'dart:math' as math;

import 'package:angular_components/angular_components'

    as angular_components;

import 'package:js/js.dart' as js;

import 'dart:math' as Math;

import 'package:angular_components/angular_components'

    as angularComponents;

import 'package:js/js.dart' as JS;
```

Nombra otros identificadores usando lowerCamelCase.

Los miembros de la clase, las definiciones de nivel superior, las variables, los parámetros y los parámetros con nombre deben escribir en mayúscula la primera letra de cada palabra, *excepto* la primera palabra, y no deben usar separadores.

```
var count = 3;

 HttpRequest httpRequest;

 void align(bool clearItems) {

  // ...

}
```

SE PREFIERE usar lowerCamelCase para nombres de constantes.

En el código nuevo, utilice lowerCamelCase para variables constantes, incluidos los valores de enumeración.

```
const pi = 3.14;

const defaultTimeout = 1000;

final urlScheme = RegExp('^([a-z]+):');


class Dice {

  static final numberGenerator = Random();

}

const PI = 3.14;

const DefaultTimeout = 1000;

final URL_SCHEME = RegExp('^([a-z]+):');


class Dice {

  static final NUMBER_GENERATOR = Random();

}
```

Puede utilizarlo SCREAMING_CAPS para mantener la coherencia con el código existente, como en los siguientes casos:

- Al agregar código a un archivo o biblioteca que ya usa SCREAMING_CAPS.
- Al generar código Dart que es paralelo al código Java

DEBE usar mayúsculas en los acrónimos y abreviaturas de más de dos letras como palabras.

Los acrónimos en mayúscula pueden ser difíciles de leer y varios acrónimos adyacentes pueden dar lugar a nombres ambiguos. Por ejemplo, dado un nombre que comienza con HTTPSFTP, no hay forma de saber si se refiere a HTTPS FTP o HTTP SFTP.

Para evitar esto, los acrónimos y abreviaturas se escriben con mayúscula como palabras normales.

Excepción: dos letras *acrónimos* como IO (entrada / salida) están completamente capitalizan: `IO`. Por otro lado, de dos letras *abreviaturas* como ID (identificación) están siendo capitalizadas como palabras regulares: `Id`.

```
class HttpConnection {}

class DBIOPort {}

class TVVcr {}

class MrRogers {}


var httpRequest = ...

var uiHandler = ...

Id id;

class HTTPConnection {}

class DbIoPort {}

class TvVcr {}

class MRRogers {}


var hTTPRequest = ...

var uIHandler = ...

ID iD;
```

Prefiero usar _, __, etc. para los parámetros de devolución de llamada no utilizados.

A veces, la firma de tipo de una función de devolución de llamada requiere un parámetro, pero la implementación de devolución de llamada no *usa* el parámetro. En este caso, es idiomático nombrar el parámetro no utilizado _. Si la función tiene varios parámetros no utilizados, utilizar guiones adicionales para evitar conflictos de nombres: __, ___, etc.

```
futureOfVoid.then((_) {

  print('Operation complete.');

});
```

Esta guía es solo para funciones que son *anónimas y locales* . Por lo general, estas funciones se usan de inmediato en un contexto en el que está claro qué representa el parámetro no utilizado. Por el contrario, las funciones de nivel superior y las declaraciones de métodos no tienen ese contexto, por lo que sus parámetros deben tener un nombre para que quede claro para qué sirve cada parámetro, incluso si no se usa.

NO use un guión bajo inicial para los identificadores que no son privados.

Dart utiliza un guion bajo inicial en un identificador para marcar a los miembros y las declaraciones de nivel superior como privados. Esto capacita a los usuarios para que asocien un subrayado inicial con uno de esos tipos de declaraciones. Ven "_" y piensan "privado".

No existe el concepto de "privado" para variables locales, parámetros, funciones locales o prefijos de biblioteca. Cuando uno de esos tiene un nombre que comienza con un guion bajo, envía una señal confusa al lector. Para evitar eso, no use guiones bajos iniciales en esos nombres.

NO use letras de prefijo.

La notación húngara y otros esquemas surgieron en la época de BCPL, cuando el compilador no hizo mucho para ayudarlo a comprender su código. Dado que Dart puede indicarle el tipo, el alcance, la mutabilidad y otras propiedades de sus declaraciones, no hay razón para codificar esas propiedades en los nombres de los identificadores.

```
defaultTimeout

kDefaultTimeout
```

Ordenando

Para mantener ordenado el preámbulo de su archivo, tenemos un orden prescrito en el que deben aparecer las directivas. Cada “sección” debe estar separada por una línea en blanco.

Una sola regla de linter maneja todas las pautas de pedido: directives_ordering.

DEBE colocar las importaciones de "flutter:" antes que otras importaciones.

```
import 'dart:async';

import 'dart:html';



import 'package:bar/bar.dart';

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

SÍ coloque las importaciones de "paquete:" antes que las importaciones relativas.

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

import 'package:foo/foo.dart';



import 'util.dart';
```

DEBE especificar las exportaciones en una sección separada después de todas las importaciones.

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

import 'src/foo_bar.dart';



export 'src/error.dart';

import 'src/error.dart';

export 'src/error.dart';

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

SÍ ordene las secciones alfabéticamente.

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

import 'package:foo/foo.dart';


import 'foo.dart';

import 'foo/foo.dart';

import 'package:foo/foo.dart';

import 'package:bar/bar.dart';


import 'foo/foo.dart';

import 'foo.dart';
```

Formateo

Como muchos idiomas, Dart/Flutter ignora los espacios en blanco. Sin embargo, los *humanos* no lo hacen. Tener un estilo de espacios en blanco consistente ayuda a garantizar que los lectores humanos vean el código de la misma manera que lo hace el compilador.

Formatee su código usando dart format.

El formateo es un trabajo tedioso y requiere mucho tiempo durante la refactorización. Afortunadamente, no tienes que preocuparte por eso. Proporcionamos un sofisticado formateador de código automatizado llamado dart format que lo hace por usted. Las reglas oficiales de manejo de espacios en blanco para Dart son *las que produce* *dart format* .

Las pautas de formato restantes son para las pocas cosas dart format que no se pueden arreglar por usted.

CONSIDERE cambiar su código para hacerlo más amigable para el formateador.

El formateador hace lo mejor que puede con cualquier código que le arrojes, pero no puede hacer milagros. Si su código tiene identificadores particularmente largos, expresiones profundamente anidadas, una mezcla de diferentes tipos de operadores, etc., la salida formateada aún puede ser difícil de leer.

Cuando eso suceda, reorganice o simplifique su código. Considere acortar el nombre de una variable local o convertir una expresión en una nueva variable local. En otras palabras, realice los mismos tipos de modificaciones que haría si formateara el código a mano y tratara de hacerlo más legible. Piense dart format en una asociación en la que trabajan juntos, a veces de forma iterativa, para producir un código atractivo.

EVITE las líneas de más de 80 caracteres.

Los estudios de legibilidad muestran que las líneas largas de texto son más difíciles de leer porque su ojo tiene que viajar más lejos cuando se mueve al principio de la siguiente línea. Ésta es la razón por la que los periódicos y las revistas utilizan varias columnas de texto.

Si realmente desea líneas de más de 80 caracteres, nuestra experiencia es que su código probablemente sea demasiado detallado y podría ser un poco más compacto.

El principal infractor suele ser VeryLongCamelCaseClassNames. Pregúntese: "¿Cada palabra en ese tipo de nombre me dice algo crítico o previene una colisión de nombres?" Si no es así, considere omitirlo.

Tenga en cuenta que dart format hace el 99% de esto por usted, pero el último 1% es usted. No divide literales de cadena larga para que quepan en 80 columnas, por lo que debe hacerlo manualmente.

Excepción: cuando aparece un URI o una ruta de archivo en un comentario o una cadena (generalmente en una importación o exportación), puede permanecer completo incluso si hace que la línea supere los 80 caracteres. Esto facilita la búsqueda de una ruta en los archivos de origen.

Excepción: las cadenas de varias líneas pueden contener líneas de más de 80 caracteres porque las nuevas líneas son importantes dentro de la cadena y dividir las líneas en líneas más cortas puede alterar el programa.

USE llaves para todas las declaraciones de control de flujo.

Si lo hace, evita el problema de colgar el resto .

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

Excepción: cuando tiene una declaración `if` sin la cláusula `else` y toda la declaración `if` cabe en una línea, puede omitir las llaves si lo prefiere:

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

Sin embargo, si el cuerpo pasa a la siguiente línea, use llaves:

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

Fuente:

https://dart.dev/guides/language/effective-dart/style