

CÉSAR VEGA LIZARAZO

Introducción al lenguaje **Dart**

Dart Language Tour
by dartlang.org CC By 3.0

Lenguaje Dart



Dart



Introducción al Lenguaje Dart

Este documento muestra cómo utilizar cada una de las principales características de Dart, desde variables y operadores hasta clases y bibliotecas, asumiendo que ya sabes cómo programar en otro lenguaje.

Este libro es resultado de la traducción de [A Tour of the Dart Language](#) en [dartlang.org](#)



Open Source
Comunidad de Flutter de
habla hispana.
G Coding Academy 2019



Traductor: César A. Vega L.

<https://twitter.com/esFlutter>

<https://medium.com/comunidad-flutter>

<https://www.facebook.com/groups/flutter.dart.spanish>

https://t.me/flutter_dart_spanish

<https://www.youtube.com/channel/UCTFLrFf1QIhiH1R1C4yq9mw>

<https://www.gcoding.academy/>

Publicado en abril de 2019

CC BY 3.0 2018

Tabla de contenido

Introducción al lenguaje Dart	8
Un programa básico de Dart.....	8
Nota: el código de este sitio sigue las convenciones de la guía de estilo de Dart.	9
Conceptos importantes	9
Palabras clave	10
Variables	11
Nota: Esta página sigue la recomendación de la guía de estilo de usar var, en lugar de anotaciones de tipo, para variables locales.....	12
Valor por defecto	12
Final y const.....	12
Tipos incorporados	14
Números.....	14
Strings	16
Booleans	18
Lists	19
Conjuntos	20
Maps.....	21
Runas.....	23
Symbols.....	24
Funciones.....	25
Dart es un verdadero lenguaje orientado a objetos, por lo que incluso las funciones son objetos y tienen un tipo, Function. Esto significa que las funciones se pueden asignar a variables o pasar como argumentos a otras funciones. También puede llamar a una instancia de una clase de Dart como si fuera una función. Para más detalles, véase Clases Invocables.	25
Parámetros opcionales.....	26
Parámetros nombrados opcionales.....	26
Parámetros de posición opcionales	27
Valores por defecto de los parámetros	27
La función main ()	29
Funciones como objetos de primera clase	29
Funciones anónimas	30
Alcance léxico.....	31

Closures Lexical	32
Funciones de pruebas para la igualdad	33
Valores de retorno	33
Operadores	34
Operadores aritméticos	35
Igualdad y operadores relacionales	36
Operadores tipo test.....	37
Operadores de asignación	38
Operadores lógicos	39
Operadores de bits y de desplazamiento	39
Expresiones condicionales	40
Notación en cascada (...)	41
Otros operadores.....	42
Sentencias de control de flujo	42
Puedes controlar el flujo de tu código Dart usando cualquiera de los siguientes:	42
If y else	43
For loops	43
While y do-while	44
Break y continue	45
Switch y case	45
Assert	47
Excepciones.....	48
Throw	48
Catch	49
Finally	51
Clases.....	51
Usando miembros de la clase	52
Uso de constructores	52
Obtener el tipo de un objeto.....	54
Variables de instancia	54
Constructores	55
Constructores predeterminados	56
Los constructores no son hereditarios	56

Constructores nombrados	56
Invocar a un constructor de superclases que no sea el predeterminado	57
Lista inicializadora.....	58
Reorientación de los constructores	59
Constructores constantes	59
Constructores de fábricas	60
Métodos	61
Métodos de instancia.....	61
Getters y setters.....	61
Métodos abstractos	62
Clases abstractas.....	63
Interfaces implícitas.....	63
Ampliación de una clase.....	64
Sobreescribir Miembros.....	65
Operadores sobreescribibles.....	65
NoSuchMethod()	66
Tipos enumerados	67
Uso de enums	67
Agregando características a una clase: mixins	68
Variables y métodos de clase	69
Variables estáticas.....	69
Métodos estáticos.....	70
Genéricos	71
¿Por qué usar genéricos?.....	71
Usando literales de la colección.....	72
Uso de tipos parametrizados con constructores.....	72
Colecciones genéricas y los tipos que contienen	73
Restricción del tipo parametrizado	73
Usando métodos genéricos.....	74
Bibliotecas y visibilidad.....	74
Uso de bibliotecas	75
Especificando un prefijo de biblioteca	75
Importar sólo una parte de una biblioteca.....	76

Cargando una biblioteca sin prisa (Lazily).....	76
Implementación de bibliotecas	77
Soporte de asincronía	77
Manejo de Futuros.....	78
Declarar funciones asíncronas.....	79
Manejo de Streams	79
Generadores	81
Clases invocables.....	82
Isolates.....	82
Typedefs	83
Metadatos.....	84
Comentarios	85
Comentarios de una sola línea	85
Comentarios de varias líneas	86
Comentarios sobre la documentación.....	86
Resumen	87
Version 2.3.0, optimizada para la construcción de interfaces de usuario.....	88
Código de ejemplo Dart utilizando el operador de propagación	88
Código de ejemplo Dart usando la colección for	89
Nuevos sitios web de Dart & Pub	89
En Dev.XX.0.....	90
Cambios en la biblioteca del core	90
dart:async.....	90
dart:core	90
dart:isolate	90
Herramientas	91
Linter.....	91
2.3.0.....	91
Lenguaje.....	91
Spread	91
Colección if	92
Colección for.....	93
Cambios en la biblioteca Core.....	94

dart:isolate	94
dart:core	94
Dart VM.....	95
Dart para la web.....	95
dart2js.....	95
Herramientas	96
Dartfmt	96
Linter.....	96
Cliente pub	96
Dart nativo	96

Introducción al lenguaje Dart

Para obtener más información sobre las bibliotecas principales de Dart, consulta [La Visita Guiada a las Bibliotecas de Dart](#). Siempre que desees más detalles sobre una característica del lenguaje, consulta la [especificación de lenguaje de Dart](#).

Consejo: Puedes jugar con la mayoría de las funciones del lenguaje usando DartPad (más información). Abre DartPad

Un programa básico de Dart

El siguiente código utiliza muchas de las funciones más básicas de Dart:

```
// Define una función.
printInteger(int aNumber) {
  print('The number is $aNumber.');// Imprime a consola.
}

// Aquí es donde la aplicación comienza a ejecutarse.
main() {
  var number = 42; // Declarar e inicializar una variable.
  printInteger(number); // Llama la función.
}
```

Esto es lo que utiliza este programa que se aplica a todas (o casi todas) las aplicaciones de Dart:

`// Esto es un comentario.`

Un comentario de una sola línea. Dart también soporta comentarios de varias líneas y documentos. Para más detalles, véase [Comentarios](#).

`int`

Un tipo. Algunos de los otros [tipos incorporados](#) son `String`, `List`, y `bool`.

`42`

Un número literal. Los literales numéricos son una especie de constante de compilación.

```
print()
```

Una forma práctica de mostrar la salida.

```
'cadena de palabras delimitadas por comilla sencilla'
```

```
(or "cadena de palabras delimitadas por comilla doble")
```

Una cadena literal.

```
$variableName (or ${expression})
```

Interpolación de strings: incluye una variable o el equivalente de una cadena de expresiones dentro de un literal de strings. Para obtener más información, véase [Strings](#).

```
main()
```

La función de alto nivel, especial, *requerida*, donde comienza la ejecución de la aplicación. Para obtener más información, consulte [la función main\(\)](#).

```
Var
```

Una forma de declarar una variable sin especificar su tipo.

Nota: el código de este sitio sigue las convenciones de la guía de estilo de Dart.

Conceptos importantes

A medida que aprendas sobre el lenguaje de Dart, ten en cuenta estos datos y conceptos:

- Todo lo que se puede colocar en una variable es un objeto, y cada objeto es una instancia de una clase. Los números pares, funciones y `null` son objetos. Todos los objetos heredan de la clase `Object`.
- Aunque Dart es fuertemente tipado, las anotaciones de tipo son opcionales porque Dart puede inferir tipos. En el código anterior, se infiere que `number` es de tipo `int`. Cuando quieras decir explícitamente que no se espera ningún tipo, [utiliza el tipo especial `dynamic`](#).
- Dart soporta tipos genéricos, como `List<int>` (una lista de enteros) o `List<dynamic>` (una lista de objetos de cualquier tipo).

- Dart soporta funciones de alto nivel (como `main()`), así como funciones vinculadas a una clase u objeto (métodos *estáticos* y de *instancia*, respectivamente). También puede crear funciones dentro de funciones (funciones *anidadas* o *locales*).
- Del mismo modo, Dart soporta variables de alto nivel, así como variables vinculadas a una clase u objeto (variables *estáticas* y de *instancia*). Las variables de instancia se conocen a veces como campos o propiedades.
- A diferencia de Java, Dart no tiene las palabras clave `public`, `protected` y `private`. Si un identificador comienza con un guión bajo (`_`), es privado a su biblioteca. Para obtener más información, consulta [Bibliotecas y visibilidad](#).
- Los *identificadores* pueden comenzar con una letra o guión bajo (`_`), seguido de cualquier combinación de esos caracteres más dígitos.
- Dart tiene tanto *expresiones* (que tienen valores de tiempo de ejecución) como *sentencias* (que no los tienen). Por ejemplo, la [expresión condicional](#) `condition ? expr1 : expr2` tiene un valor de `expr1` o `expr2`. Compáralo con una [sentencia if-else](#), que no tiene valor. Una sentencia a menudo contiene una o más expresiones, pero una expresión no puede contener directamente una sentencia.
- Las herramientas de Dart pueden reportar dos tipos de problemas: advertencias (warnings) y errores (errors). Las advertencias son sólo indicaciones de que tu código podría no funcionar, pero no impiden que tu programa se ejecute. Los errores pueden ser en tiempo de compilación o en tiempo de ejecución. Un error en tiempo de compilación impide que el código se ejecute en absoluto; un error en tiempo de ejecución provoca que se produzca una [excepción](#) mientras el código se ejecuta.

Palabras clave

La siguiente tabla enumera las palabras que el lenguaje Dart trata de manera especial.

<code>abstract</code> ²	<code>dynamic</code> ²	<code>implements</code> ²	<code>show</code> ¹
<code>as</code> ²	<code>else</code>	<code>import</code> ²	<code>static</code> ²
<code>assert</code>	<code>enum</code>	<code>in</code>	<code>super</code>
<code>async</code> ¹	<code>export</code> ²	<code>interface</code> ²	<code>switch</code>
<code>await</code> ³	<code>extends</code>	<code>is</code>	<code>sync</code> ¹
<code>break</code>	<code>external</code> ²	<code>library</code> ²	<code>this</code>
<code>case</code>	<code>factory</code> ²	<code>mixin</code> ²	<code>throw</code>

catch	false	new	true
class	final	null	try
const	finally	on ¹	typedef ²
continue	for	operator ²	var
covariant ²	Function ²	part ²	void
default	get ²	rethrow	while
deferred ²	hide ¹	return	with
do	if	set ²	yield ³

Evita usar estas palabras como identificadores. Sin embargo, si es necesario, las palabras clave marcadas con superíndices pueden ser identificadores:

- Las palabras con el superíndice **1** son **palabras clave contextuales**, que tienen significado sólo en lugares específicos. Son identificadores válidos en todas partes.
- Las palabras con el superíndice **2** son **identificadores incorporados**. Para simplificar la tarea de portar código JavaScript a Dart, estas palabras clave son identificadores válidos en la mayoría de los lugares, pero no se pueden usar como nombres de clase o tipo, o como prefijos de importación.
- Las palabras con el superíndice **3** son palabras reservadas más nuevas y limitadas relacionadas con el [soporte asincrónico](#) que se agregó después de la versión 1.0 de Dart. No se puede utilizar `await` o `yield` como identificador en ningún cuerpo de función marcado con `async`, `async*` o `sync*`.

Todas las demás palabras de la tabla son **palabras reservadas**, que no pueden ser identificadores.

Variables

Aquí hay un ejemplo de cómo crear una variable e inicializarla:

```
var name = 'Bob';
```

Las variables almacenan referencias. La variable llamada `name` contiene una referencia a un objeto `String` con un valor de "Bob".

El tipo de la variable `name` se deduce que es `String`, pero puede cambiar ese tipo especificándolo. Si un objeto no está restringido a un solo tipo, especifique el tipo `Object` o `dynamic`, siguiendo las [guías de diseño](#).

```
dynamic name = 'Bob';
```

Otra opción es declarar explícitamente el tipo que se inferiría:

```
String name = 'Bob';
```

Nota: Esta página sigue la [recomendación de la guía de estilo](#) de usar `var`, en lugar de anotaciones de tipo, para variables locales.

Valor por defecto

Las variables no inicializadas tienen un valor inicial `null`. Incluso las variables con tipos numéricos son inicialmente `null`, porque los números -como todo lo demás en Dart- son objetos.

```
int lineCount;  
assert(lineCount == null);
```

Nota: La llamada a `assert()` se ignora en el código de producción. Durante el desarrollo, `assert(condition)` lanza una excepción a menos que la condición sea verdadera. Para más detalles, véase `Assert`.

Final y const

Si nunca tienes la intención de cambiar una variable, utiliza `final` o `const`, ya sea en lugar de `var` o en adición a un tipo. Una variable final sólo se puede establecer una sola vez; una variable constante es una constante en tiempo de compilación (Las

variables `const` son implícitamente finales). Una variable final de alto nivel o de clase se inicializa la primera vez que se usa.

Nota: Las variables de instancia pueden ser `final` pero no `const`. Las variables de instancia final deben inicializarse antes de que comience el cuerpo del constructor: en la declaración de la variable, mediante un parámetro del constructor o en la lista de inicializadores del constructor.

He aquí un ejemplo de cómo crear y establecer una variable final:

```
final name = 'Bob'; // Sin una anotación de tipo
final String nickname = 'Bobby';
```

No puedes cambiar el valor de una variable final:

```
name = 'Alice'; // Error: una variable final solo se puede establecer una vez.
```

Usa `const` para las variables que quieras que sean constantes de tiempo de compilación. Si la variable `const` está a nivel de clase, márkela como `static const`. Cuando declare la variable, establezca el valor de una constante de tiempo de compilación, como un número o un string literal, una variable constante o el resultado de una operación aritmética sobre números constantes:

```
const bar = 1000000; // Unidad de presión (dynes/cm2)
const double atm = 1.01325 * bar; // Atmosfera Estándar
```

La palabra clave `const` no es solo para declarar variables constantes. También puede usarlo para crear *valores* constantes, así como para declarar constructores que *crean* valores constantes. Cualquier variable puede tener un valor constante.

```
var foo = const [];  
final bar = const [];  
const baz = []; // Equivalente a `const []`
```

Puedes omitir `const` de la expresión inicial de una declaración `const`, como en el caso anterior de `baz`. Para obtener más información, véase [NO use const redundantemente](#).

Puedes cambiar el valor de una variable no final, no constante, incluso si solía tener un valor `const`:

```
foo = [1, 2, 3]; // Era const []
```

No se puede cambiar el valor de una variable constante:

```
baz = [42]; // Error: Las variables constantes no pueden ser asignadas a un valor.
```

Para obtener más información sobre el uso de `const` para crear valores constantes, consulte [Lists](#), [Maps](#) y [Classes](#).

Tipos incorporados

El lenguaje Dart tiene soporte especial para los siguientes tipos:

- numbers
- strings
- booleans
- lists (también conocidos como *arrays*)
- sets
- maps
- runes (for expressing Unicode characters in a string)
- symbols

Puedes inicializar un objeto de cualquiera de estos tipos especiales utilizando un literal. Por ejemplo, `'esto es una cadena'` es un literal de strings, y `true` es un literal booleano.

Debido a que cada variable en Dart se refiere a un objeto -una instancia de una clase- normalmente puedes usar *constructores* para inicializar variables. Algunos de los tipos incorporados tienen sus propios constructores. Por ejemplo, puede utilizar el constructor `Map()` para crear un mapa.

Números

Los números de Dart vienen en dos sabores:

`int`

Valores enteros no mayores de 64 bits, dependiendo de la plataforma. En la VM de Dart, los valores pueden ser de -2^{63} a $2^{63} - 1$. Dart compilado en JavaScript usa [números JavaScript](#), permitiendo valores de -2^{53} a $2^{53} - 1$.

double

Números de coma flotante de 64 bits (doble precisión), tal y como se especifica en el estándar IEEE 754.

Tanto `int` como `double` son subtipos de `num`. El tipo `num` incluye operadores básicos como `+`, `-`, `/`, y `*`, y es también donde encontrarás `abs()`, `ceil()`, y `floor()`, entre otros métodos. (Los operadores Bitwise, como `>>`, están definidos en la clase `int`.) Si `num` y sus subtipos no tienen lo que estás buscando, la librería `dart:math` podría tenerlo.

Los números enteros son números sin punto decimal. Aquí hay algunos ejemplos de definición de literales enteros:

```
var x = 1;  
var hex = 0xDEADBEEF;
```

Si un número incluye un decimal, es un doble. Aquí hay algunos ejemplos de definición de literales double:

```
var y = 1.1;  
var exponents = 1.42e5;
```

A partir de Dart 2.1, los literales enteros se convierten automáticamente a dobles cuando es necesario:

```
double z = 1; // Equivalent to double z = 1.0.
```

Nota de la versión: antes de Dart 2.1, era un error utilizar un literal entero en un contexto doble.

Así es como se convierte un string en un número, o viceversa:

```
// String -> int
var one = int.parse('1');
assert(one == 1);

// String -> double
var onePointOne = double.parse('1.1');
assert(onePointOne == 1.1);

// int -> String
String oneAsString = 1.toString();
assert(oneAsString == '1');

// double -> String
String piAsString = 3.14159.toStringAsFixed(2);
assert(piAsString == '3.14');
```

El tipo `int` especifica los operadores tradicionales de desplazamiento binario (`<<`, `>>`), AND (`&`), y OR (`|`). Por ejemplo:

```
assert((3 << 1) == 6); // 0011 << 1 == 0110
assert((3 >> 1) == 1); // 0011 >> 1 == 0001
assert((3 | 4) == 7); // 0011 | 0100 == 0111
```

Los números literales son constantes en tiempo de compilación. Muchas expresiones aritméticas son también constantes en tiempo de compilación, siempre y cuando sus operandos sean constantes en tiempo de compilación que evalúen a números.

```
const msPerSecond = 1000;
const secondsUntilRetry = 5;
const msUntilRetry = secondsUntilRetry * msPerSecond;
```

Strings

Un string de Dart es una secuencia de unidades de código UTF-16. Puedes usar comillas simples o dobles para crear un string:

```
var s1 = 'Single quotes work well for string literals.';
var s2 = "Double quotes work just as well.";
var s3 = 'It\'s easy to escape the string delimiter.';
var s4 = "It's even easier to use the other delimiter.";
```

Puedes poner el valor de una expresión dentro de un string usando `${expression}`. Si la expresión es un identificador, puedes omitir el `{}`. Para obtener el string correspondiente a un objeto, Dart llama al método `toString()` del objeto.

```
var s = 'string interpolation';

assert('Dart has $s, which is very handy.' ==
      'Dart has string interpolation, ' +
      'which is very handy.');
```

```
assert('That deserves all caps. ' +
      '${s.toUpperCase()} is very handy!' ==
      'That deserves all caps. ' +
      'STRING INTERPOLATION is very handy!');
```

Nota: El operador `==` comprueba si dos objetos son equivalentes. Dos strings son equivalentes si contienen la misma secuencia de unidades de código.

Puedes concatenar strings utilizando literales de string adyacentes o el operador `+`:

```
var s1 = 'String '
        'concatenation'
        " works even over line breaks.";
assert(s1 ==
      'String concatenation works even over '
      'line breaks.');
```

```
var s2 = 'The + operator ' + 'works, as well.';
assert(s2 == 'The + operator works, as well.');
```

Otra forma de crear un string multilínea: utilice una comilla triple con comillas simples o dobles:

```
var s1 = '''
Puedes crear
multilíneas de strings como éste.
''';
```

```
var s2 = """Esta es también un
string multilinea.""";
```

Puede crear un string "raw" añadiéndole el prefijo r:

```
var s = r'In a raw string, not even \n gets special treatment.';
```

Consulta [Runas](#) para obtener más información sobre cómo expresar caracteres Unicode en un string.

Los strings literales son constantes en tiempo de compilación, siempre y cuando cualquier expresión interpolada sea una constante en tiempo de compilación que se evalúe como nula o un valor numérico, de string, o booleano.

```
// Estos trabajan en un string const.
const aConstNum = 0;
const aConstBool = true;
const aConstString = 'a constant string';

// Estos NO trabajan en un string const.
var aNum = 0;
var aBool = true;
var aString = 'a string';
const aConstList = [1, 2, 3];

const validConstString = '$aConstNum $aConstBool $aConstString';
// const invalidConstString = '$aNum $aBool $aString $aConstList';
```

Para obtener más información sobre el uso de strings, consulta [Strings y expresiones regulares](#).

Booleans

Para representar valores booleanos, Dart tiene un tipo llamado `bool`. Solo dos objetos tienen el tipo `bool`: los literales booleanos `true` y `false`, que son constantes en tiempo de compilación.

El tipo en Dart seguramente significa que no se puede usar código como `if (nonbooleanValue)` o `assert (nonbooleanValue)`. En su lugar, verifique explícitamente los valores, de esta manera:

```
// Comprobar si hay un string vacío.
var fullName = '';
assert(fullName.isEmpty);

// Comprobar por cero.
var hitPoints = 0;
assert(hitPoints <= 0);

// Comprobar por null.
var unicorn;
assert(unicorn == null);

// Comprobar por NaN.
var iMeantToDoThis = 0 / 0;
assert(iMeantToDoThis.isNaN);
```

Lists

Quizás la colección más común en casi todos los lenguajes de programación es el array, o grupo ordenado de objetos. En Dart, los arrays son objetos [List](#), así que la mayoría de la gente los llama *listas*.

Los literales de lista de Dart se parecen a los del array de JavaScript. Aquí hay una simple lista Dart:

```
var list = [1, 2, 3];
```

Nota: Dart deduce que `list` tiene el tipo `List<int>`. Si intentas agregar objetos no enteros a esta lista, el analizador o el runtime generarán un error. Para más información, lee acerca de la [inferencia de tipos](#).

Las listas utilizan la indexación basada en cero, donde 0 es el índice del primer elemento y `list.length - 1` es el índice del último elemento. Puedes obtener la longitud de una lista y referirte a los elementos de la lista tal como lo harías en JavaScript:

```
var list = [1, 2, 3];
assert(list.length == 3);
assert(list[1] == 2);
```

```
list[1] = 1;
assert(list[1] == 1);
```

Para crear una lista que sea constante en tiempo de compilación, agrega `const` antes del literal de lista:

```
var constantList = const [1, 2, 3];
// constantList[1] = 1; // Si se descomenta esto, se produce un error.
```

El tipo de lista tiene muchos métodos útiles para manipular listas. Para obtener más información acerca de las listas, consulte [Genéricos](#) y [colecciones](#).

Conjuntos

Un Set en Dart es una colección desordenada de elementos únicos. El soporte de Dart para conjuntos se proporciona mediante literales de conjuntos y el tipo [Set](#).

Nota de la versión: Aunque el *tipo* Set siempre ha sido una parte central de Dart, los conjuntos de *literales* se introdujeron en Dart 2.2.

Aquí hay un conjunto de Dart simple, creado usando un conjunto de literales:

```
var halogens = {'fluorine', 'chlorine', 'bromine', 'iodine', 'astatine'};
```

Nota: Dart deduce que los `halogens` tienen el tipo `Set<String>`. Si intentas agregar un tipo de valor incorrecto al conjunto, el analizador o el runtime generarán un error. Para más información, lee acerca de la [inferencia de tipos](#).

Para crear un conjunto vacío, use `{}` precedido por un argumento de tipo, o asigne `{}` a una variable de tipo `Set`:

```
var names = <String>{};
// Set<String> names = {}; // Esto trabaja, también.
// var names = {}; // Crea un map, no un set.
```

¿Set o map? La sintaxis para los literales de map es similar a la de los literales de set. Debido a que los literales de map son los primeros, {} predetermina el tipo `Map`. Si olvidas la anotación de tipo {} o la variable a la que está asignada, entonces Dart crea un objeto de tipo `Map<dynamic, dynamic>`.

Agrega elementos a un conjunto existente usando los métodos `add()` o `addAll()`:

```
var elements = <String>{};
elements.add('fluorine');
elements.addAll(halogens);
```

Use `.length` para obtener el número de elementos en el conjunto:

```
var elements = <String>{};
elements.add('fluorine');
elements.addAll(halogens);
assert(elements.length == 5);
```

Para crear un conjunto que sea constante en tiempo de compilación, agregue `const` antes del literal set:

```
final constantSet = const {
  'fluorine',
  'chlorine',
  'bromine',
  'iodine',
  'astatine',
};
// constantSet.add('helium'); // Descomentar esto, causa un error.
```

Para obtener más información sobre los conjuntos, consulte [Generics](#) y [Sets](#).

Maps

En general, un mapa (map) es un objeto que asocia claves (keys) y valores (values). Tanto las claves como los valores pueden ser cualquier tipo de objeto. Cada clave se produce una sola vez, pero se puede utilizar el mismo valor varias veces. El soporte de Dart para los mapas es proporcionado por los literales de mapas y el tipo de `Map`.

Aquí hay un par de mapas simples de Dart, creados con literales de mapas:

```
var gifts = {
  // Key:    Value
  'first': 'partridge',
  'second': 'turtledoves',
  'fifth': 'golden rings'
};

var nobleGases = {
  2: 'helium',
  10: 'neon',
  18: 'argon',
};
```

Nota: Dart infiere que `gifts` tiene el tipo `Map<String, String>` y `nobleGases` tiene el tipo `Map<int, String>`. Si intentas agregar un tipo de valor incorrecto a cualquiera de los mapas, el analizador o el tiempo de ejecución generarán un error. Para más información, lee acerca de la [inferencia de tipos](#)

Puedes crear los mismos objetos usando un constructor de mapas:

```
var gifts = Map();
gifts['first'] = 'partridge';
gifts['second'] = 'turtledoves';
gifts['fifth'] = 'golden rings';

var nobleGases = Map();
nobleGases[2] = 'helium';
nobleGases[10] = 'neon';
nobleGases[18] = 'argon';
```

Nota: Es posible que esperes ver un `new Map()` en lugar de solo un `Map()`. A partir de Dart 2, la palabra clave `new` es opcional. Para más detalles, ver [Uso de constructores](#)

Agrega un nuevo par clave-valor a un mapa existente como lo haría en JavaScript:

```
var gifts = {'first': 'partridge'};
gifts['fourth'] = 'calling birds'; // Agregaa una pareja key-value
```

Recupera un valor de un mapa de la misma manera que lo harías en JavaScript:

```
var gifts = {'first': 'partridge'};
assert(gifts['first'] == 'partridge');
```

Si buscas una clave que no está en un mapa, obtendrás un null a cambio:

```
var gifts = {'first': 'partridge'};
assert(gifts['fifth'] == null);
```

Usa `.length` para obtener el número de pares clave-valor en el mapa:

```
var gifts = {'first': 'partridge'};
gifts['fourth'] = 'calling birds';
assert(gifts.length == 2);
```

Para crear un mapa que sea una constante en tiempo de compilación, agrega `const` antes del literal del mapa:

```
final constantMap = const {
  2: 'helium',
  10: 'neon',
  18: 'argon',
};

// constantMap[2] = 'Helium'; // Descomentar esto causa un error.
```

Para obtener más información sobre los conjuntos, consulte [Genéricos](#) y [Maps](#).

Runas

En Dart, las runas son los puntos de código UTF-32 de un string.

Unicode define un valor numérico único para cada letra, dígito y símbolo utilizado en todos los sistemas de escritura del mundo. Debido a que un string de Dart es una secuencia de unidades de código UTF-16, la expresión de valores Unicode de 32 bits dentro de un string requiere una sintaxis especial.

La forma habitual de expresar un punto de código Unicode es `\uXXXX`, donde XXXX es un valor hexadecimal de 4 dígitos. Por ejemplo, el personaje del corazón (♥) es

\u2665. Para especificar más o menos de 4 dígitos hexadecimales, coloca el valor entre corchetes. Por ejemplo, el emoji sonriente (😊) es \u{1f600}.

La clase `String` tiene varias propiedades que puedes usar para extraer información de las runas. Las propiedades `codeUnitAt` y `codeUnit` devuelven unidades de código de 16 bits. Usa la propiedad `runes` para obtener las `runes` de un string.

El siguiente ejemplo ilustra la relación entre las runas, las unidades de código de 16 bits y los puntos de código de 32 bits.

```
main() {
  var clapping = '\u{1f44f}';
  print(clapping);
  print(clapping.codeUnits);
  print(clapping.runes.toList());

  Runes input = new Runes(
    '\u2665 \u{1f605} \u{1f60e} \u{1f47b} \u{1f596} \u{1f44d}');
  print(new String.fromCharCode(input));
}
```

😊
[55357, 56399]
[128079]

♥ 😊 😎 🐼 🙌 👍

Nota: Ten cuidado al manipular runas utilizando operaciones de lista. Este enfoque puede descomponerse fácilmente, según el lenguaje, el conjunto de caracteres y la operación en particular. Para obtener más información, consulta [¿Cómo puedo revertir un String en Dart?](#) en un Stack Overflow.

Symbols

Un objeto `Symbol` representa un operador o identificador declarado en un programa de Dart. Puede que nunca necesites utilizar símbolos, pero son muy valiosos para las API que se refieren a identificadores por nombre, ya que la minificación cambia los nombres de los identificadores, pero no los símbolos de identificación.

Para obtener el símbolo de un identificador, use un símbolo literal, que es `#` seguido del identificador:

```
#radix
#bar
```

Los literales de símbolos son constantes en tiempo de compilación.

Funciones

Dart es un verdadero lenguaje orientado a objetos, por lo que incluso las funciones son objetos y tienen un tipo, [Function](#). Esto significa que las funciones se pueden asignar a variables o pasar como argumentos a otras funciones. También puede llamar a una instancia de una clase de Dart como si fuera una función. Para más detalles, véase [Clases Invocables](#).

He aquí un ejemplo de implementación de una función:

```
bool isNoble(int atomicNumber) {  
  return _nobleGases[atomicNumber] != null;  
}
```

Aunque Effective Dart recomienda [anotaciones de tipo para APIs públicas](#), la función sigue funcionando si omite los tipos:

```
isNoble(atomicNumber) {  
  return _nobleGases[atomicNumber] != null;  
}
```

Para las funciones que contienen una sola expresión, puedes utilizar una sintaxis abreviada:

```
bool isNoble(int atomicNumber) => _nobleGases[atomicNumber] != null;
```

La sintaxis `=> expr` es una abreviatura de `{return expr; }`. La notación `=>` a veces se denomina sintaxis de *flecha*.

Nota: Solo puede aparecer una expresión, no una sentencia, entre la flecha (`=>`) y el punto y coma (`;`). Por ejemplo, no puedes poner una [instrucción if](#) allí, pero puede usar una [expresión condicional](#)

Una función puede tener dos tipos de parámetros: requeridos y opcionales. Los parámetros requeridos se enumeran primero, seguidos de los parámetros opcionales. Los parámetros opcionales nombrados también se pueden marcar como `@required`. Véase la siguiente sección para más detalles.

Parámetros opcionales

Los parámetros opcionales pueden ser posicionales o nombrados, pero no ambos.

Parámetros nombrados opcionales

Al llamar a una función, puede especificar parámetros con nombre usando `paramName: value`. Por ejemplo:

```
enableFlags(bold: true, hidden: false);
```

Al definir una función, usa `{param1, param2, ...}` para especificar parámetros nombrados:

```
/// Establece las banderas [bold] y [hidden] ...  
void enableFlags({bool bold, bool hidden}) {...}
```

Las expresiones de creación de instancias de [Flutter](#) ([inglés](#)) pueden volverse complejas, por lo que los constructores de widgets utilizan exclusivamente parámetros nombrados. Esto hace que las expresiones de creación de instancias sean más fáciles de leer.

Puedes anotar un parámetro nombrado en cualquier código de Dart (no sólo Flutter) con `@required` para indicar que es un parámetro *requerido*. Por ejemplo:

```
const Scrollbar({Key key, @required Widget child})
```

Cuando se construye un `Scrollbar`, el analizador informa de un problema cuando el argumento `child` está ausente.

Requerido se define en el paquete **meta**. Puede importar `package:meta/meta.dart` directamente, o importar otro paquete que exporte **meta**, como el paquete de `Flutter:flutter/material.dart`.

Parámetros de posición opcionales

Al envolver un conjunto de parámetros de función en `[]` se marcan como parámetros de posición opcionales:

```
String say(String from, String msg, [String device]) {
  var result = '$from says $msg';
  if (device != null) {
    result = '$result with a $device';
  }
  return result;
}
```

Aquí hay un ejemplo de cómo llamar a esta función sin el parámetro opcional:

```
assert(say('Bob', 'Howdy') == 'Bob says Howdy');
```

Y aquí hay un ejemplo de llamar a esta función con el tercer parámetro:

```
assert(say('Bob', 'Howdy', 'smoke signal') ==
  'Bob says Howdy with a smoke signal');
```

Valores por defecto de los parámetros

Su función puede utilizar `=` para definir valores por defecto tanto para los parámetros nombrados como para los parámetros posicionales. Los valores por defecto deben ser constantes en tiempo de compilación. Si no se proporciona ningún valor por defecto, el valor por defecto es `null`.

A continuación, se muestra un ejemplo de configuración de valores predeterminados para los parámetros nombrados:

```
/// Establece las banderas [bold] y [hidden] ...
void enableFlags({bool bold = false, bool hidden = false}) {...}

// bold será true; hidden será false.
enableFlags(bold: true);
```

Nota de depreciación: El código antiguo puede usar dos puntos (:) en lugar de = para establecer los valores predeterminados de los parámetros nombrados. La razón es que originalmente, sólo se admitía: para los parámetros nombrados. Es probable que este soporte sea obsoleto, por lo que le recomendamos que **utilice = para especificar los valores por defecto**.

El siguiente ejemplo muestra cómo establecer valores predeterminados para parámetros posicionales:

```
String say(String from, String msg,
  [String device = 'carrier pigeon', String mood]) {
  var result = '$from says $msg';
  if (device != null) {
    result = '$result with a $device';
  }
  if (mood != null) {
    result = '$result (in a $mood mood)';
  }
  return result;
}

assert(say('Bob', 'Howdy') ==
  'Bob says Howdy with a carrier pigeon');
```

También puede pasar listas o mapas como valores predeterminados. El siguiente ejemplo define una función, `doStuff()`, que especifica una lista por defecto para el parámetro `list` y un mapa por defecto para el parámetro `gifts`.

```
void doStuff(
  {List<int> list = const [1, 2, 3],
   Map<String, String> gifts = const {
     'first': 'paper',
     'second': 'cotton',
     'third': 'leather'
   }}) {
  print('list: $list');
  print('gifts: $gifts');
}
```


La función main ()

Cada aplicación debe tener una función `main()` de alto nivel, que sirve como punto de entrada a la aplicación. La función `main()` devuelve vacío y tiene un parámetro opcional `List<String>` para los argumentos.

He aquí un ejemplo de la función `main()` para una aplicación web:

```
void main() {
  querySelector('#sample_text_id')
    ..text = 'Click me!'
    ..onClick.listen(reverseText);
}
```

Nota: La sintaxis de `..` en el código anterior se llama *cascada*. Con las cascadas, puede realizar múltiples operaciones en los miembros de un solo objeto.

Aquí hay un ejemplo de la función `main()` para una aplicación de línea de comandos que toma argumentos:

```
// Ejecuta la aplicación de esta manera: dart args.dart 1 test
void main(List<String> arguments) {
  print(arguments);

  assert(arguments.length == 2);
  assert(int.parse(arguments[0]) == 1);
  assert(arguments[1] == 'test');
}
```

Puedes usar la [biblioteca args](#) para definir y analizar argumentos de línea de comandos.

Funciones como objetos de primera clase

Puedes pasar una función como parámetro a otra función. Por ejemplo:

```
void printElement(int element) {
  print(element);
}
```

```
var list = [1, 2, 3];

// Pasa printElement como parámetro.
list.forEach(printElement);
```

También puede asignar una función a una variable, como, por ejemplo:

```
var loudify = (msg) => '!!! ${msg.toUpperCase()} !!!';
assert(loudify('hello') == '!!! HELLO !!!');
```

Este ejemplo utiliza una función anónima. Más información en la siguiente sección.

Funciones anónimas

La mayoría de las funciones tienen nombre, como `main()` o `printElement()`. También puede crear una función sin nombre llamada función anónima, o a veces una lambda o closure. Puede asignar una función anónima a una variable para que, por ejemplo, pueda añadirla o eliminarla de una colección.

Una función anónima se parece a una función nombrada - cero o más parámetros, separados por comas y anotaciones de tipo opcional, entre paréntesis.

El bloque de código que sigue contiene el cuerpo de la función:

```
([[Type] param1[, ...]]) {
  codeBlock;
};
```

El siguiente ejemplo define una función anónima con un parámetro sin tipo, `item`. La función, invocada para cada elemento de la lista, imprime un string que incluye el valor en el índice especificado.

```
var list = ['apples', 'bananas', 'oranges'];
list.forEach((item) {
  print('${list.indexOf(item)}: $item');
});
```

Ejecución del código.

```
void main() {  
  var list = ['apples', 'bananas', 'oranges'];  
  list.forEach((item) {  
    print('${list.indexOf(item)}: $item');  
  });  
}
```

0: apples
1: bananas
2: oranges

Si la función solo contiene una declaración, puede acortarla utilizando la notación de flecha. Pegua la siguiente línea en DartPad y haz clic en ejecutar para verificar que es funcionalmente equivalente.

```
list.forEach(  
  (item) => print('${list.indexOf(item)}: $item'));
```

Alcance léxico

Dart es un lenguaje de alcance léxico, lo que significa que el alcance de las variables se determina estáticamente, simplemente por la disposición del código. Puede "seguir las llaves hacia afuera" para ver si hay una variable en el alcance.

A continuación, se muestra un ejemplo de funciones anidadas con variables en cada nivel de alcance:

```
bool topLevel = true;  
  
void main() {  
  var insideMain = true;  
  
  void myFunction() {  
    var insideFunction = true;  
  
    void nestedFunction() {  
      var insideNestedFunction = true;
```

```

    assert(topLevel);
    assert(insideMain);
    assert(insideFunction);
    assert(insideNestedFunction);
  }
}
}

```

Note cómo `nestedFunction()` puede usar variables desde cada nivel, hasta el nivel superior.

Closures Lexical

Un closure es un objeto de función que tiene acceso a variables en su ámbito léxico, incluso cuando la función se utiliza fuera de su ámbito original.

Las funciones pueden cerrar sobre variables definidas en los alcances circundantes. En el siguiente ejemplo, `makeAdder()` captura la variable `addBy`. Donde quiera que vaya la función devuelta, recuerda `addBy`.

```

/// Retorna una función que agrega [addBy] al argumento
/// de la función.
Function makeAdder(num addBy) {
  return (num i) => addBy + i;
}

void main() {
  // Create a function that adds 2.
  var add2 = makeAdder(2);

  // Create a function that adds 4.
  var add4 = makeAdder(4);

  assert(add2(3) == 5);
  assert(add4(3) == 7);
}

```

Funciones de pruebas para la igualdad

Aquí hay un ejemplo de prueba de funciones de alto nivel, métodos estáticos y métodos de instancia para la igualdad:

```
void foo() {} // Una función de alto nivel

class A {
  static void bar() {} // A static method
  void baz() {} // An instance method
}

void main() {
  var x;

  // Comparando función de alto nivel.
  x = foo;
  assert(foo == x);

  // Comparando métodos estáticos.
  x = A.bar;
  assert(A.bar == x);

  // Comparando métodos de instancia.
  var v = A(); // Instance #1 of A
  var w = A(); // Instance #2 of A
  var y = w;
  x = w.baz;

  // Esos closures refieren a diferentes instancias,
  // así que son desiguales.
  assert(v.baz != w.baz);
}
```

Valores de retorno

Todas las funciones devuelven un valor. Si no se especifica ningún valor de retorno, la instrucción `return null`; Se adjunta implícitamente al cuerpo de la función.

```
foo() {}

assert(foo() == null);
```

Operadores

Dart define los operadores que se muestran en la siguiente tabla. Puedes anular muchos de estos operadores, como se describe en [Operadores sobrescribibles](#).

Descripción	Operador
unary postfix	<i>expr</i> ++ <i>expr</i> -- () [] . ?.
unary prefix	- <i>expr</i> ! <i>expr</i> ~ <i>expr</i> ++ <i>expr</i> -- <i>expr</i>
multiplicative	* / % ~/
additive	+ -
shift	<< >> >>>
bitwise AND	&
bitwise XOR	^
bitwise OR	
relational and type test	>= > <= < as is is!
equality	== !=
logical AND	&&
logical OR	
if null	??
conditional	<i>expr1</i> ? <i>expr2</i> : <i>expr3</i>
cascade	..
assignment	= *= /= += -= &= ^= etc.

Advertencia: la precedencia del operador es una aproximación del comportamiento de un analizador de Dart. Para obtener respuestas definitivas, consulta la gramática en la [especificación del lenguaje Dart](#).

Cuando usas operadores, creas expresiones. Aquí hay algunos ejemplos de expresiones de operador:

```
a++
a + b
a = b
a == b
c ? a : b
a is T
```

En la tabla de operadores, cada operador tiene mayor precedencia que los operadores de las filas siguientes. Por ejemplo, el operador multiplicativo % tiene

mayor precedencia que (y por lo tanto ejecuta antes) el operador de igualdad `==`, que tiene mayor precedencia que el operador lógico AND `&&`. Esa precedencia significa que las siguientes dos líneas de código se ejecutan de la misma manera:

```
// Los paréntesis mejoran la legibilidad.
if ((n % i == 0) && (d % i == 0)) ...

// Más difícil de leer, pero equivalente.
if (n % i == 0 && d % i == 0) ...
```

Advertencia: Para los operadores que trabajan en dos operandos, el operando situado más a la izquierda determina qué versión del operador se usa. Por ejemplo, si tienes un objeto `Vector` y un objeto `Point`, `aVector + aPoint` usa la versión `Vector` de `+`.

Operadores aritméticos

Dart soporta los operadores aritméticos habituales, como se muestra en la siguiente tabla.

Operador	Significado
<code>+</code>	Adicionar
<code>-</code>	Subtraer
<code>-expr</code>	Unary menos, también conocido como negación (invierte el signo de la expresión)
<code>*</code>	Multiplicación
<code>/</code>	División
<code>~/</code>	Divide, devolviendo un resultado entero
<code>%</code>	Obtener el resto de una división entera (módulo)

Ejemplo:

```
assert(2 + 3 == 5);
assert(2 - 3 == -1);
assert(2 * 3 == 6);
assert(5 / 2 == 2.5); // Result is a double
assert(5 ~/ 2 == 2); // Result is an int
assert(5 % 2 == 1); // Remainder

assert('5/2 = ${5 ~/ 2} r ${5 % 2}' == '5/2 = 2 r 1');
```


Dart también admite operadores de incremento y decremento tanto de prefijo como de postfijo.

Operador	Significado
<code>++var</code>	<code>var = var + 1</code> (valor expresión es <code>var + 1</code>)
<code>var++</code>	<code>var = var + 1</code> (valor expresión es <code>var</code>)
<code>--var</code>	<code>var = var - 1</code> (valor expresión es <code>var - 1</code>)
<code>var--</code>	<code>var = var - 1</code> (valor expresión es <code>var</code>)

Ejemplo:

```
var a, b;

a = 0;
b = ++a; // Incremento a antes que b obtenga su valor.
assert(a == b); // 1 == 1

a = 0;
b = a++; // Incremento a DESPUÉS que b obtenga su valor.
assert(a != b); // 1 != 0

a = 0;
b = --a; // Decremento a antes que b obtenga su valor.
assert(a == b); // -1 == -1

a = 0;
b = a--; // Decremento a DESPUÉS que b obtenga su valor.
assert(a != b); // -1 != 0
```

Igualdad y operadores relacionales

En el siguiente cuadro se enumeran los significados de la igualdad y de los operadores relacionales.

Operador	Significado
<code>==</code>	Igual; ver discusión abajo
<code>!=</code>	No igual
<code>></code>	Mayor que
<code><</code>	Menor que
<code>>=</code>	Mayor o igual que
<code><=</code>	Menor o igual que

Para comprobar si dos objetos `x` y `y` representan la misma cosa, utiliza el operador `==`. (En el raro caso de que necesites saber si dos objetos son exactamente el mismo objeto, usa la función `identical()` en su lugar.) Así es como funciona el operador `==`:

1. Si `x` o `y` es nulo, devuelve `true` si ambos son `null`, y `false` si sólo uno es `null`.
2. Devuelve el resultado de la invocación del método `x.==(y)`. (Así es, operadores como `==` son métodos que se invocan en su primer operando. Incluso puedes sobrescribir muchos operadores, incluyendo `==`, como verás en [Operadores Sobrescribibles](#)).

He aquí un ejemplo del uso de cada uno de los operadores de igualdad y relacionales:

```
assert(2 == 2);
assert(2 != 3);
assert(3 > 2);
assert(2 < 3);
assert(3 >= 3);
assert(2 <= 3);
```

Operadores tipo test

Los operadores `as`, `is`, e `is!` son útiles para verificar los tipos en tiempo de ejecución.

Operador	Significado
<code>as</code>	Typecast (también se utiliza para especificar los prefijos de la biblioteca)
<code>is</code>	True si el objeto tiene el tipo especificado
<code>is!</code>	False si el objeto no tiene el tipo especificado

El resultado de `obj is T` es `true` si `obj` implementa la interfaz especificada por `T`. Por ejemplo, `obj is Object` es siempre `true`.

Usa el operador `as` para proyectar un objeto a un tipo en particular. En general, deberías usarlo como abreviatura para una prueba de `is` sobre un objeto seguido de una expresión que utilice ese objeto. Por ejemplo, considera el siguiente código:

```
if (emp is Person) {
  // Type check
  emp.firstName = 'Bob';
}
```

Puedes acortar el código usando el operador `as`:

```
(emp as Person).firstName = 'Bob';
```

Nota: el código no es equivalente. Si `emp` es nulo o no es un `Person`, el primer ejemplo (con `is`) no hace nada; el segundo (con `as`) lanza una excepción.

Operadores de asignación

Como ya has visto, puedes asignar valores usando el operador `=`. Para asignar sólo si la variable asignada es null, utiliza el operador `??=`.

```
// Asignar valor a "a"
a = value;
// Asignar valor a "b" if "b" es null; de lo contrario, "b" permanece
igual
b ??= value;
```

Los operadores de asignación compuesta como `+=` combinan una operación con una asignación.

<code>=</code>	<code>-=</code>	<code>/=</code>	<code>%=</code>	<code>>>=</code>	<code>^=</code>
<code>+=</code>	<code>*=</code>	<code>~/=</code>	<code><<=</code>	<code>&=</code>	<code> =</code>

Aquí es cómo funcionan los operadores de asignación compuesta:

	Asignación compuesta	Expresión equivalente
Para un operador <code>op</code> :	<code>a op= b</code>	<code>a = a op b</code>
Ejemplo:	<code>a += b</code>	<code>a = a + b</code>

El siguiente ejemplo utiliza operadores de asignación y asignación compuesta:

```
var a = 2; // Asignar utilizando =
a *= 3; // Asignar y multiplicar: a = a * 3
assert(a == 6);
```

Operadores lógicos

Puedes invertir o combinar expresiones booleanas utilizando los operadores lógicos.

Operador	Significado
<code>!expr</code>	invierte la siguiente expresión (cambia falso a verdadero y viceversa)
<code> </code>	OR lógico
<code>&&</code>	AND lógico

Aquí hay un ejemplo del uso de los operadores lógicos:

```
if (!done && (col == 0 || col == 3)) {
  // ... Hacer algo...
}
```

Operadores de bits y de desplazamiento

Puedes manipular los bits individuales de los números en Dart. Por lo general, se utilizan estos operadores de bits y de desplazamiento con enteros.

Operador	Significado
<code>&</code>	AND
<code> </code>	OR
<code>^</code>	XOR
<code>~expr</code>	Unary bitwise complement (0s become 1s; 1s become 0s)
<code><<</code>	Shift left
<code>>></code>	Shift right

He aquí un ejemplo del uso de operadores de bits y de desplazamiento:

```
final value = 0x22;
final bitmask = 0x0f;

assert((value & bitmask) == 0x02); // AND
assert((value & ~bitmask) == 0x20); // AND NOT
assert((value | bitmask) == 0x2f); // OR
assert((value ^ bitmask) == 0x2d); // XOR
assert((value << 4) == 0x220); // Shift left
assert((value >> 4) == 0x02); // Shift right
```

Expresiones condicionales

Dart tiene dos operadores que le permiten evaluar de manera concisa expresiones que de otra manera podrían requerir declaraciones `if-else`:

`condition ? expr1 : expr2`

Si la condición es verdadera, evalúa `expr1` (y devuelve su valor); de lo contrario, evalúa y devuelve el valor de `expr2`.

`expr1 ?? expr2`

Si `expr1` no es nulo, devuelve su valor; de lo contrario, evalúa y devuelve el valor de `expr2`.

Cuando requieras asignar un valor basado en una expresión booleana, considera el uso de `?:`.

```
var visibility = isPublic ? 'public' : 'private';
```

Si la expresión booleana prueba para null, considera usar `??`.

```
String playerName(String name) => name ?? 'Guest';
```

El ejemplo anterior podría haberse escrito al menos de otras dos maneras, pero no tan sucintamente:

```
// En versiones ligeramente más largas usa el operador ?:
String playerName(String name) => name != null ? name : 'Guest';

// En versiones muy largas usa la sentencia if-else.
String playerName(String name) {
  if (name != null) {
```

```

    return name;
  } else {
    return 'Guest';
  }
}

```

Notación en cascada (...)

Las cascadas (..) permiten realizar una secuencia de operaciones sobre el mismo objeto. Además de las llamadas de función, también se puede acceder a los campos de ese mismo objeto. Esto a menudo te ahorra el paso de crear una variable temporal y te permite escribir un código más fluido.

Considera el siguiente código:

```

querySelector('#confirm') // Obtiene un objeto.
..text = 'Confirm' // Usa sus miembros.
..classes.add('important')
..onClick.listen((e) => window.alert('Confirmed!'));

```

La primera llamada al método, `querySelector()`, devuelve un objeto selector. El código que sigue a la notación en cascada opera sobre este objeto selector, ignorando cualquier valor posterior que pueda ser devuelto.

El ejemplo anterior es equivalente a:

```

var button = querySelector('#confirm');
button.text = 'Confirm';
button.classes.add('important');
button.onClick.listen((e) => window.alert('Confirmed!'));

```

También puedes anidar tus cascadas. Por ejemplo:

```

final addressBook = (AddressBookBuilder()
  ..name = 'jenny'
  ..email = 'jenny@example.com'
  ..phone = (PhoneNumberBuilder()
    ..number = '415-555-0100'
    ..label = 'home')
    .build())
  .build();

```

Ten cuidado de construir tu cascada en una función que devuelva un objeto real. Por ejemplo, el siguiente código falla:

```
var sb = StringBuffer();
sb.write('foo')
  ..write('bar'); // Error: método 'write' no está definido por 'void'.
```

La llamada `sb.write()` devuelve `void`, y no se puede construir una cascada en `void`.

Nota: Estrictamente hablando, la notación de "punto doble" para las cascadas no es un operador. Es solo parte de la sintaxis de Dart.

Otros operadores

Has visto la mayoría de los operadores restantes en otros ejemplos:

Operador	Nombre	Significado
<code>()</code>	Aplicación de funciones	Representa el llamado a una función
<code>[]</code>	Lista de acceso	Se refiere al valor en el índice especificado en la lista
<code>.</code>	Acceso de miembros	Se refiere a una propiedad de una expresión; ejemplo: <code>foo.bar</code> selecciona la propiedad <code>bar</code> de la expresión <code>foo</code>
<code>?.</code>	Acceso condicional para miembros	Como <code>.</code> , Pero el operando más a la izquierda puede ser nulo; ejemplo: <code>foo? .bar</code> selecciona la propiedad <code>bar</code> de la expresión <code>foo</code> a menos que <code>foo</code> sea nulo (en cuyo caso el valor de <code>foo? .bar</code> es nulo)

Para obtener más información sobre los operadores `.`, `?.`, y `..`, consulte [Clases](#).

Sentencias de control de flujo

Puedes controlar el flujo de tu código Dart usando cualquiera de los siguientes:

- `if` y `else`
- `for` loops
- `while` y `do-while` loops
- `break` y `continue`
- `switch` y `case`
- `assert`

También puede afectar el flujo de control utilizando `try-catch` y `throw`, como se explica en [Excepciones](#).

If y else

Dart soporta sentencias `if` con sentencias `else` opcionales, como se muestra en el siguiente ejemplo. Ver también [expresiones condicionales](#).

```
if (isRaining()) {
  you.bringRainCoat();
} else if (isSnowing()) {
  you.wearJacket();
} else {
  car.putTopDown();
}
```

A diferencia de JavaScript, las condiciones deben usar valores booleanos, nada más. Ver [Booleans](#) para más información.

For loops

Puedes iterar con el estándar del bucle `for`. Por ejemplo:

```
var message = StringBuffer('Dart is fun');
for (var i = 0; i < 5; i++) {
  message.write('!');
}
```

Los Closures dentro de los bucles `for` de Dart capturan el *valor* del índice, evitando un error común que se encuentra en JavaScript. Por ejemplo, considere:

```
var callbacks = [];
```



```
for (var i = 0; i < 2; i++) {
  callbacks.add(() => print(i));
}
callbacks.forEach((c) => c());
```

La salida es `0` y luego `1`, como se esperaba. En contraste, el ejemplo imprimiría `2` y luego `2` en JavaScript.

Si el objeto sobre el que está iterando es un Iterable, puede usar el método `forEach()`. Usar `forEach()` es una buena opción si no necesitas conocer el contador de iteración actual:

```
candidates.forEach((candidate) => candidate.interview());
```

Las clases iterables, como List y Set, también soportan la forma de **iteración for-in**:

```
var collection = [0, 1, 2];
for (var x in collection) {
  print(x); // 0 1 2
}
```

While y do-while

Un bucle `while` evalúa la condición antes del bucle:

```
while (!isDone()) {
  doSomething();
}
```

Un bucle `do-while` evalúa la condición después del bucle:

```
do {
  printLine();
} while (!atEndOfPage());
```

Break y continue

Usa `break` para detener el bucle:

```
while (true) {
  if (shutdownRequested()) break;
  processIncomingRequests();
}
```

Utiliza `Continue` para saltar a la siguiente iteración de bucle:

```
for (int i = 0; i < candidates.length; i++) {
  var candidate = candidates[i];
  if (candidate.yearsExperience < 5) {
    continue;
  }
  candidate.interview();
}
```

Puedes escribir ese ejemplo de manera diferente si estás utilizando un `Iterable` como una lista o conjunto:

```
candidates
  .where((c) => c.yearsExperience >= 5)
  .forEach((c) => c.interview());
```

Switch y case

Las sentencias Switch en Dart comparan números enteros, cadenas o constantes en tiempo de compilación usando `==`. Los objetos comparados deben ser todas instancias de la misma clase (y no de ninguno de sus subtipos), y la clase no debe sobrescribir `==`. Los `tipos numerados` funcionan bien en las sentencias `switch`.

Nota: las sentencias Switch en Dart están pensadas para circunstancias limitadas, como en intérpretes o escáneres.

Cada cláusula de `case` no vacía termina con una declaración `break`, como regla general. Otras formas de validar el fin de una cláusula `case` no vacía, es una sentencia `continue`, `throw`, o `return`.

Utilice una cláusula `default` para ejecutar código cuando no coincida con ninguna cláusula `case`:

```
var command = 'OPEN';
switch (command) {
  case 'CLOSED':
    executeClosed();
    break;
  case 'PENDING':
    executePending();
    break;
  case 'APPROVED':
    executeApproved();
    break;
  case 'DENIED':
    executeDenied();
    break;
  case 'OPEN':
    executeOpen();
    break;
  default:
    executeUnknown();
}
```

El siguiente ejemplo omite la sentencia `break` en una cláusula `case`, generando así un error:

```
var command = 'OPEN';
switch (command) {
  case 'OPEN':
    executeOpen();
    // ERROR: Falta de break

  case 'CLOSED':
    executeClosed();
    break;
}
```

Sin embargo, Dart soporta cláusulas `case`, permitiendo una forma de pase a través de ella:

```
var command = 'CLOSED';
switch (command) {
  case 'CLOSED': // Case vacío pase a través de él.
  case 'NOW_CLOSED':
    // Ejecuta para ambos CLOSED y NOW_CLOSED.
    executeNowClosed();
    break;
}
```

Si realmente deseas un pase a través de él, puede usar una sentencia `continue` y una etiqueta:

```
var command = 'CLOSED';
switch (command) {
  case 'CLOSED':
    executeClosed();
    continue nowClosed;
    // Continúa ejecutando en el label nowClosed.

  nowClosed:
  case 'NOW_CLOSED':
    // Se ejecuta para CLOSED y NOW_CLOSED.
    executeNowClosed();
    break;
}
```

Una cláusula `case` puede tener variables locales, que son visibles solo dentro del alcance de esa cláusula.

Assert

Utiliza una sentencia `assert` para interrumpir la ejecución normal si una condición booleana es falsa. Puedes encontrar ejemplos de declaraciones de afirmación a lo largo de esta guía. Aquí hay más:

```
// Asegúrate de que la variable tiene un valor no nulo.
assert(text != null);

// Asegúrate de que el valor sea menor que 100.
assert(number < 100);

// Asegúrate de que esta es una URL https.
```

```
assert(urlString.startsWith('https'));
```

Nota: Las sentencias Assert no tienen ningún efecto en el código de producción; son sólo para desarrollo. Flutter permite realizar afirmaciones en [modo debug](#). Las herramientas de sólo desarrollo como [dartdevc](#) normalmente soportan asserts por defecto. Algunas herramientas, como [dart](#) y [dart2js](#), soportan asserts a través de una bandera en la línea de comandos: `--enable-asserts`.

Para adjuntar un mensaje a una assert, agrega una cadena como segundo argumento.

```
assert(urlString.startsWith('https'),
      'URL ($urlString) should start with "https".');
```

El primer argumento de `assert` puede ser cualquier expresión que se resuelva a un valor booleano. Si el valor de la expresión es verdadero, el assert tiene éxito y la ejecución continúa. Si es falsa, el assert falla y se lanza una excepción (un [AssertionError](#)).

Excepciones

Tu código de Dart puede lanzar y atrapar excepciones. Las excepciones son los errores que indican que algo inesperado sucedió. Si la excepción no es capturada, el isolate que levantó la excepción es suspendido, y típicamente el isolate y su programa son terminados.

A diferencia de Java, todas las excepciones de Dart son excepciones no verificadas. Los métodos no declaran qué excepciones pueden lanzar, y no se requiere que capturen ninguna excepción.

Dart proporciona tipos de [Excepción](#) y [Error](#), así como numerosos subtipos predefinidos. Por supuesto, puedes definir tus propias excepciones. Sin embargo, los programas de Dart pueden lanzar cualquier objeto no nulo, no sólo objetos de Excepción y Error, como excepción.

Throw

Aquí hay un ejemplo de lanzar o *levantar* una excepción:

```
throw FormatException('Expected at least 1 section');
```

También puedes lanzar objetos arbitrarios:

```
throw 'Out of llamas!';
```

Note: El código de calidad de producción usualmente arroja tipos que implementan `Error` o `Excepción`.

Debido a que lanzar una excepción es una expresión, puede lanzar excepciones en sentencias =>, así como en cualquier otro lugar que permita expresiones:

```
void distanceTo(Point other) => throw UnimplementedError();
```

Catch

Catching, o capturar, una excepción impide que la excepción se propague (a menos que vuelva a emitir la excepción). Atrapar una excepción te da la oportunidad de manejarlo:

```
try {
  breedMoreLlamas();
} on OutOfLlamasException {
  buyMoreLlamas();
}
```

Para manejar código que puede lanzar más de un tipo de excepción, puedes especificar múltiples cláusulas de captura. La primera cláusula de captura que coincide con el tipo de objeto lanzado maneja la excepción. Si la cláusula de captura no especifica un tipo, esa cláusula puede manejar cualquier tipo de objeto lanzado:

```
try {
  breedMoreLlamas();
} on OutOfLlamasException {
  // Una excepción específica
  buyMoreLlamas();
} on Exception catch (e) {
  // Cualquier otra cosa que sea una excepción.
  print('Unknown exception: $e');
} catch (e) {
```

```
// Ningún tipo especificado, maneja todos
print('Something really unknown: $e');
}
```

Como muestra el código anterior, puedes usar `on` o `catch` o ambos. Usa `on` cuando necesites especificar el tipo de excepción. Usa `catch` cuando tu manejador de excepciones necesite el objeto de excepción.

Puedes especificar uno o dos parámetros para `catch()`. El primero es la excepción que fue lanzada, y el segundo es el rastro de pila (un objeto `StackTrace`).

```
try {
  // ...
} on Exception catch (e) {
  print('Exception details:\n $e');
} catch (e, s) {
  print('Exception details:\n $e');
  print('Stack trace:\n $s');
}
```

Para manejar parcialmente una excepción, mientras te permite propagarse, usa la palabra clave `rethrow`.

```
void misbehave() {
  try {
    dynamic foo = true;
    print(foo++); // Error de Runtime
  } catch (e) {
    print('misbehave() partially handled ${e.runtimeType}.');
    rethrow; // Permitir a los invocadores ver la excepción.
  }
}

void main() {
  try {
    misbehave();
  } catch (e) {
    print('main() finished handling ${e.runtimeType}.');
  }
}
```

Finally

Para asegurarse de que algún código se ejecuta independientemente de que se haya lanzado o no una excepción, utiliza una cláusula `finally`. Si ninguna cláusula `catch` coincide con la excepción, la excepción se propaga después de que se ejecute la cláusula `finally`:

```
try {
  breedMoreLlamas();
} finally {
  // Always clean up, even if an exception is thrown.
  cleanLlamaStalls();
}
```

La cláusula `finally` se ejecuta después de cualquier cláusula `catch` coincidente:

```
try {
  breedMoreLlamas();
} catch (e) {
  print('Error: $e'); // Manejar la excepción primero.
} finally {
  cleanLlamaStalls(); // Entonces limpia.
}
```

Para conocer más, lee la sección [Excepciones](#) de la visita guiada de la biblioteca.

Clases

Dart es un lenguaje orientado a objetos con clases y herencia basada en mixin (mezclas). Cada objeto es una instancia de una clase, y todas las clases descienden de [Object](#). La *herencia basada en Mixin* significa que, aunque cada clase (excepto por `Object`) tiene exactamente una superclase, un cuerpo de clase puede ser reutilizado en múltiples jerarquías de clases.

Usando miembros de la clase

Los objetos tienen *miembros* que consisten en funciones y datos (*métodos* y *variables de instancia*, respectivamente). Cuando se *invoca* a un método, se *invoca* a un objeto: el método tiene acceso a las funciones y datos de ese objeto.

Utilice un punto (.) para referirse a una variable o método de instancia:

```
var p = Point(2, 2);

// Establecer el valor de la variable de instancia y.
p.y = 3;

// Obtener el valor de y.
assert(p.y == 3);

// Invocar distanceTo () en p.
num distance = p.distanceTo(Point(4, 4));
```

Usa `?.` en lugar de `.` para evitar una excepción cuando el operando más a la izquierda es nulo:

```
// Si p no es nulo, establezca su valor y en 4.
p?.y = 4;
```

Uso de constructores

Puedes crear un objeto utilizando un *constructor*. Los nombres de los constructores pueden ser `ClassName` o `ClassName.identifier`. Por ejemplo, el siguiente código crea objetos `Point` usando los constructores `Point()` y `Point.fromJson()`:

```
var p1 = new Point(2, 2);
var p2 = new Point.fromJson({'x': 1, 'y': 2});
```

El siguiente código tiene el mismo efecto, pero usa la palabra clave `new` opcional antes del nombre del constructor:

```
var p1 = new Point(2, 2);
var p2 = new Point.fromJson({'x': 1, 'y': 2});
```

Nota de la versión: la palabra clave `new` se convirtió en opcional en Dart 2

Algunas clases proporcionan **constructores constantes**. Para crear una constante en tiempo de compilación usando un constructor de constantes, ponga la palabra clave `const` antes del nombre del constructor:

```
var p = const ImmutablePoint(2, 2);
```

La construcción de dos constantes idénticas en tiempo de compilación da como resultado una instancia canónica única:

```
var a = const ImmutablePoint(1, 1);
var b = const ImmutablePoint(1, 1);

assert(identical(a, b)); // ¡Son la misma instancia!
```

Dentro de un *contexto constante*, puede omitir `const` antes de un constructor o literal. Por ejemplo, mira este código, que crea un mapa de constante:

```
// Un montón de palabras clave const aquí.
const pointAndLine = const {
  'point': const [const ImmutablePoint(0, 0)],
  'line': const [const ImmutablePoint(1, 10), const ImmutablePoint(-2,
11)],
};
```

Puedes omitir todo excepto el primer uso de la palabra clave `const`:

```
// Sólo una constante, que establece el contexto constante.
const pointAndLine = {
```

```
'point': [ImmutablePoint(0, 0)],
'line': [ImmutablePoint(1, 10), ImmutablePoint(-2, 11)],
};
```

Si un constructor constante está fuera de un contexto constante y se invoca sin `const`, crea un **objeto no constante**:

```
var a = const ImmutablePoint(1, 1); // Crea una constante
var b = ImmutablePoint(1, 1); // No crea una constante
assert(!identical(a, b)); // No son la misma instancia!
```

Nota de la versión: la palabra clave `const` se convirtió en opcional en un contexto constante en Dart 2.

Obtener el tipo de un objeto

Para obtener el tipo de un objeto en tiempo de ejecución, puede usar la propiedad `runtimeType` de `Object`, que devuelve un objeto `Type`.

```
print('The type of a is ${a.runtimeType}');
```

Hasta aquí has visto cómo *usar* las clases. El resto de esta sección muestra cómo *implementar* clases.

Variables de instancia

Aquí vemos la forma en la que se declaran las variables de instancia:

```
class Point {
  num x; // Declarar la variable de instancia x, inicialmente nula.
  num y; // Declara y, inicialmente nula.
  num z = 0; // Declara z, inicialmente 0.
}
```

Todas las variables de instancia no inicializadas tienen el valor `null`.

Todas las variables de instancia generan un método *getter* implícito. Las variables de instancia no finales también generan un método *setter* implícito. Para más detalles, consulta [Getters y setters](#).

```
class Point {
  num x;
  num y;
}

void main() {
  var point = Point();
  point.x = 4; // Use the setter method for x.
  assert(point.x == 4); // Use the getter method for x.
  assert(point.y == null); // Values default to null.
}
```

Si inicializas una variable de instancia donde se declara (en lugar de en un constructor o método), el valor se establece cuando se crea la instancia, que es antes de que se ejecuten el constructor y su lista de inicializadores.

Constructores

Declarar un constructor creando una función con el mismo nombre que su clase (más, opcionalmente, un identificador adicional como se describe en [Constructores Nombrados](#)). La forma más común de constructor, el constructor generativo, crea una nueva instancia de una clase:

```
class Point {
  num x, y;

  Point(num x, num y) {
    // There's a better way to do this, stay tuned.
    this.x = x;
    this.y = y;
  }
}
```

La palabra clave `this` se refiere a la instancia actual.

Nota: utiliza `this` solo cuando exista algún conflicto de nombres. De lo contrario, el estilo Dart omite `this`.

El patrón de asignar un argumento de constructor a una variable de instancia es muy común, Dart tiene azúcar sintáctica para facilitarlo:

```
class Point {
  num x, y;

  // Syntactic sugar for setting x and y
  // before the constructor body runs.
  Point(this.x, this.y);
}
```

Constructores predeterminados

Si no declaras un constructor, se te proporciona un constructor predeterminado. El constructor por defecto no tiene argumentos e invoca al constructor sin argumentos de la superclase.

Los constructores no son hereditarios

Las subclases no heredan constructores de su superclase. Una subclase que declara que no hay constructores sólo tiene el constructor predeterminado (sin argumento, sin nombre).

Constructores nombrados

Utilice un constructor nombrado para implementar múltiples constructores para una clase o para proporcionar claridad adicional:

```
class Point {
  num x, y;

  Point(this.x, this.y);

  // Constructor nombrado
  Point.origin() {
    x = 0;
    y = 0;
  }
}
```

Recuerda que los constructores no son heredados, lo que significa que el constructor nombrado de una superclase no es heredado por una subclase. Si deseas que una subclase se cree con un constructor con nombre definido en la superclase, debes implementar ese constructor en la subclase.

Invocar a un constructor de superclases que no sea el predeterminado

Por defecto, un constructor en una subclase llama al constructor sin nombre de la superclase, sin argumentos. El constructor de la superclase es llamado al principio del cuerpo del constructor. Si también se está utilizando una [lista de inicializadores](#), se ejecuta antes de que se llame a la superclase. En resumen, el orden de ejecución es el siguiente:

1. lista de inicializadores
2. constructor no-arg de superclases
3. constructor no-arg de la clase main

Si la superclase no tiene un constructor sin nombre y sin argumentos, entonces debe llamar manualmente a uno de los constructores de la superclase. Especifique el constructor de la superclase después de dos puntos (:), justo antes del cuerpo del constructor (si lo hay).

En el siguiente ejemplo, el constructor de la clase Employee llama al constructor nombrado para su superclase, Person.

```
class Person {
  String firstName;

  Person.fromJson(Map data) {
    print('in Person');
  }
}

class Employee extends Person {
  // Person does not have a default constructor;
  // you must call super.fromJson(data).
  Employee.fromJson(Map data) : super.fromJson(data) {
    print('in Employee');
  }
}

main() {
  var emp = new Employee.fromJson({});
}
```

in Person
in Employee

Debido a que los argumentos del constructor de la superclase se evalúan antes de invocar al constructor, un argumento puede ser una expresión, así como una llamada a la función:

```
class Employee extends Person {
  Employee() : super.fromJson(getDefaultData());
  // ...
}
```

Advertencia: los argumentos al constructor de la superclase no tienen acceso a `this`. Por ejemplo, los argumentos pueden llamar a métodos estáticos, pero no a métodos de instancia.

Lista inicializadora

Además de invocar a un constructor de la superclase, también puedes inicializar las variables de instancia antes de que se ejecute el cuerpo del constructor. Separa los inicializadores con comas.

```
// La lista de inicializadores establece las variables de instancia
// antes de que el cuerpo del constructor se ejecute.
Point.fromJson(Map<String, num> json)
  : x = json['x'],
    y = json['y'] {
  print('In Point.fromJson(): ($x, $y)');
}
```

Advertencia: el lado derecho de un inicializador no tiene acceso a `this`.

Durante el desarrollo, puedes validar entradas utilizando `assert` en la lista de inicializadores.

```
Point.withAssert(this.x, this.y) : assert(x >= 0) {
  print('In Point.withAssert(): ($x, $y)');
}
```

Las listas de inicialización son útiles al configurar los campos finales. El siguiente ejemplo inicializa tres campos finales en una lista de inicializadores.

```
import 'dart:math';

class Point {
  final num x;
  final num y;
  final num distanceFromOrigin;

  Point(x, y)
    : x = x,
      y = y,
      distanceFromOrigin = sqrt(x * x + y * y);
}

main() {
  var p = new Point(2, 3);
  print(p.distanceFromOrigin);
}
```

3.605551275463989

Reorientación de los constructores

A veces, el único propósito de un constructor es redirigir a otro constructor en la misma clase. El cuerpo del constructor que redirige es vacío, con la llamada del constructor apareciendo después de dos puntos (:).

```
class Point {
  num x, y;

  // El constructor principal para esta clase.
  Point(this.x, this.y);

  // Delegados al constructor principal.
  Point.alongXAxis(num x) : this(x, 0);
}
```

Constructores constantes

Si tu clase produce objetos que nunca cambian, puedes hacer que estos objetos sean constantes en tiempo de compilación. Para hacer esto, define un constructor `const` y asegúrate de que todas las variables de instancia sean `final`.

```
class ImmutablePoint {
  static final ImmutablePoint origin =
    const ImmutablePoint(0, 0);
}
```



```

    final num x, y;

    const ImmutablePoint(this.x, this.y);
}

```

Los constructores constantes no siempre crean constantes. Para más detalles, vea la sección sobre el [uso de constructores](#).

Constructores de fábricas

Utiliza la palabra clave `factory` al implementar un constructor que no siempre crea una nueva instancia de su clase. Por ejemplo, un constructor `factory` puede devolver una instancia de una caché, o puede devolver una instancia de un subtipo.

El siguiente ejemplo muestra un constructor de fábrica que devuelve objetos de una caché:

```

class Logger {
    final String name;
    bool mute = false;

    // _cache es biblioteca privada, gracias al _
    // delante de su nombre.
    static final Map<String, Logger> _cache =
        <String, Logger>{};

    factory Logger(String name) {
        if (_cache.containsKey(name)) {
            return _cache[name];
        } else {
            final logger = Logger._internal(name);
            _cache[name] = logger;
            return logger;
        }
    }

    Logger._internal(this.name);

    void log(String msg) {
        if (!mute) print(msg);
    }
}

```

Nota: Los constructores de fábricas no tienen acceso a `this`.

Invoca un constructor de fábrica como lo harías con cualquier otro constructor:

```
var logger = Logger('UI');  
logger.log('Button clicked');
```

Métodos

Los métodos son funciones que proporcionan comportamiento a un objeto.

Métodos de instancia

Los métodos de instancia sobre objetos pueden acceder a las variables de instancia y `this`. El método `distanceTo()` en la siguiente muestra es un ejemplo de un método de instancia:

```
import 'dart:math';  
  
class Point {  
  num x, y;  
  
  Point(this.x, this.y);  
  
  num distanceTo(Point other) {  
    var dx = x - other.x;  
    var dy = y - other.y;  
    return sqrt(dx * dx + dy * dy);  
  }  
}
```

Getters y setters

Los Getters y Setters son métodos especiales que proporcionan acceso de lectura y escritura a las propiedades de un objeto. Recuerda que cada variable de instancia tiene un getter implícito, más un setter si es apropiado. Puedes crear propiedades adicionales implementando getters y setters, usando las palabras clave `get` y `set`:

```
class Rectangle {  
  num left, top, width, height;  
  
  Rectangle(this.left, this.top, this.width, this.height);  
}
```

```
// Definir dos propiedades calculadas: right y bottom.
num get right => left + width;
set right(num value) => left = value - width;
num get bottom => top + height;
set bottom(num value) => top = value - height;
}

void main() {
  var rect = Rectangle(3, 4, 20, 15);
  assert(rect.left == 3);
  rect.right = 12;
  assert(rect.left == -8);
}
```

Con los getters y los setters, puedes comenzar con las variables de instancia, luego envolverlas con métodos, todo sin cambiar el código del cliente.

Nota: Los operadores como increment (++) funcionan de la forma esperada, ya sea que se defina explícitamente o no un getter. Para evitar efectos secundarios inesperados, el operador llama al getter exactamente una vez, guardando su valor en una variable temporal.

Métodos abstractos

Los métodos de instancia getter y setter pueden ser abstractos definiendo una interfaz, pero dejando su implementación a otras clases. Los métodos abstractos sólo pueden existir en [clases abstractas](#).

Para hacer un método abstracto, utilice un punto y coma (;) en lugar de un cuerpo de método:

```
abstract class Doer {
  // Define variables de instancia y métodos...

  void doSomething(); // Define un método abstracto.
}

class EffectiveDoer extends Doer {
  void doSomething() {
    // Proporciona una implementación, por lo que el método
    // no es abstracto aquí...
  }
}
```

Clases abstractas

Usa el modificador `abstract` para definir una clase abstracta, una clase que no puede ser instanciada. Las clases abstractas son útiles para definir interfaces, a menudo con alguna implementación. Si deseas que tu clase abstracta parezca instanciable, define un [constructor factory](#).

Las clases abstractas a menudo tienen [métodos abstractos](#). He aquí un ejemplo de cómo declarar una clase abstracta que tiene un método abstracto:

```
// Su clase es declarada abstracta y por lo tanto
// o puede ser instanciada.
abstract class AbstractContainer {
  // Definir constructores, campos, métodos...

  void updateChildren(); // Abstract method.
}
```

Interfaces implícitas

Cada clase define implícitamente una interfaz que contiene todos los miembros de instancia de la clase y de las interfaces que implementa. Si deseas crear una clase A que soporte la API de la clase B sin heredar la implementación de B, la clase A debería implementar la interfaz B.

Una clase implementa una o más interfaces declarándolas en una cláusula `implements` y luego proporcionando las APIs requeridas por las interfaces. Por ejemplo:

```
// Una persona. La interfaz implícita contiene greet().
class Person {
  // En la interfaz, pero visible solo en esta librería.
  final _name;

  // No en la interfaz, ya que este es un constructor.
  Person(this._name);

  // En la interfaz.
  String greet(String who) => 'Hello, $who. I am $_name.';
}
```

```
// Una implementación de la interfaz de Person.
class Impostor implements Person {
  get _name => '';

  String greet(String who) => 'Hi $who. Do you know who I am?';
}

String greetBob(Person person) => person.greet('Bob');

void main() {
  print(greetBob(Person('Kathy')));
  print(greetBob(Impostor()));
}
```

Aquí hay un ejemplo de especificar que una clase implementa múltiples interfaces:

```
class Point implements Comparable, Location {...}
```

Ampliación de una clase

Usa `extends` para crear una subclase, y `super` para referirse a la superclase:

```
class Television {
  void turnOn() {
    _illuminateDisplay();
    _activateIrSensor();
  }
  // ...
}

class SmartTelevision extends Television {
  void turnOn() {
    super.turnOn();
    _bootNetworkInterface();
    _initializeMemory();
    _upgradeApps();
  }
  // ...
}
```

Sobreescribir Miembros

Las subclases pueden sobreescribir métodos de instancia, getters y setters. Puedes usar la anotación `@override` para indicar que estás sobreescribiendo intencionalmente a un miembro:

```
class SmartTelevision extends Television {
  @override
  void turnOn() {...}
  // ...
}
```

Para especificar el tipo de parámetro de método o variable de instancia en código que sea de **tipo seguro**, puedes usar la palabra clave `covariant`.

Operadores sobreescribibles

Puedes sobreescribir los operadores mostrados en la siguiente tabla. Por ejemplo, si defines una clase vectorial, puedes definir un método `+` para añadir dos vectores.

<	+		[]
>	/	^	[]=
<=	~/	&	~
>=	*	<<	==
-	%	>>	

Nota: Puede que hayas notado que `!=` no es un operador que se pueda sobreescribir. La expresión `e1 != e2` es sólo azúcar sintáctico para `!(e1 == e2)`.

Aquí hay un ejemplo de una clase que sobreescribe los operadores `+` y `-`:

```
class Vector {
  final int x, y;

  Vector(this.x, this.y);

  Vector operator +(Vector v) => Vector(x + v.x, y + v.y);
  Vector operator -(Vector v) => Vector(x - v.x, y - v.y);

  // El operator == y hashCode no son mostrados. Por detalles,
```

```
// mira la nota abajo.
// ...
}

void main() {
  final v = Vector(2, 3);
  final w = Vector(2, 2);

  assert(v + w == Vector(4, 5));
  assert(v - w == Vector(0, 1));
}
```

Si sobrescribe `==`, también debería sobrescribir el `hashCode` de `Object`. Para un ejemplo de sobrescritura de `==` y `hashCode`, véase [Implementación de claves de mapa](#).

Para obtener más información sobre la sustitución, en general, véase [Ampliación de una clase](#).

noSuchMethod()

Para detectar o reaccionar cuando el código intenta usar un método o variable de instancia inexistente, puede sobrescribir `noSuchMethod()`:

```
class A {
  // Unless you override noSuchMethod, using a
  // non-existent member results in a NoSuchMethodError.
  @override
  void noSuchMethod(Invocation invocation) {
    print('You tried to use a non-existent member: ' +
          '${invocation.memberName}');
  }
}
```

No se puede invocar un método no implementado a menos que **uno** de los siguientes sea verdadero:

- El receptor tiene el tipo estático `dynamic`.
- El receptor tiene un tipo estático que define el método no implementado (abstract está bien), y el tipo dinámico del receptor tiene una implementación de `noSuchMethod()` que es diferente a la de la clase `Object`.

Para obtener más información, consulta [la especificación de reenvío informal noSuchMethod](#).

Tipos enumerados

Los tipos enumerados, a menudo llamados *enumeraciones* o *enums*, son un tipo especial de clase utilizada para representar un número fijo de valores constantes.

Uso de enums

Declare un tipo enumerado usando la palabra clave `enum`:

```
enum Color { red, green, blue }
```

Cada valor en un enum tiene un getter `index`, que devuelve la posición basada en cero del valor en la declaración enum. Por ejemplo, el primer valor tiene el índice 0, y el segundo valor tiene el índice 1.

```
assert(Color.red.index == 0);  
assert(Color.green.index == 1);  
assert(Color.blue.index == 2);
```

Para obtener una lista de todos los valores de enum, usa la constante `values` de la enumeración.

```
List<Color> colors = Color.values;  
assert(colors[2] == Color.blue);
```

Puede usar enums de las [sentencias switch](#), y recibirás una advertencia si no manejas todos los valores de la enumeración:

```
var aColor = Color.blue;  
  
switch (aColor) {  
  case Color.red:  
    print('Red as roses!');  
    break;  
  case Color.green:  
    print('Green as grass!');  
    break;  
}
```



```
default: // Sin esto, ves un WARNING.
  print(aColor); // 'Color.blue'
}
```

Las clases enumeradas tienen los siguientes límites:

- No puedes subclasificar, mezclar o implementar un enum.
- No puedes instanciar explícitamente una enum.

Para obtener más información, consulta la [especificación del lenguaje de Dart](#).

Agregando características a una clase: mixins

Los Mixins son una forma de reutilizar el código de una clase en múltiples jerarquías de clases.

Para utilizar un mixin, utiliza la palabra clave `with` seguida de uno o más nombres de mixin. El siguiente ejemplo muestra dos clases que utilizan mixins:

```
class Musician extends Performer with Musical {
  // ...
}

class Maestro extends Person
  with Musical, Aggressive, Demented {
  Maestro(String maestroName) {
    name = maestroName;
    canConduct = true;
  }
}
```

Para implementar una mezcla, cree una clase que extienda de `Object` y declare que no hay constructores. A menos que desees que tu mixin sea utilizable como una clase regular, usa la palabra clave `mixin` en lugar de `class`. Por ejemplo:

```
mixin Musical {
  bool canPlayPiano = false;
  bool canCompose = false;
  bool canConduct = false;

  void entertainMe() {
    if (canPlayPiano) {
      print('Playing piano');
    } else if (canConduct) {
```

```

        print('Waving hands');
    } else {
        print('Humming to self');
    }
}
}

```

Para especificar que sólo ciertos tipos pueden usar el mixin, por ejemplo, para que su mixin pueda invocar un método que no define, usa `on` para especificar la superclase requerida:

```

mixin MusicalPerformer on Musician {
    // ...
}

```

Nota de la versión: el soporte para la palabra clave `mixin` se introdujo en Dart 2.1. El código en versiones anteriores usualmente usaba `abstract class` en su lugar. Para obtener más información sobre los cambios de 2.1 en `mixin`, consulta el [registro de cambios del SDK de Dart](#) y la [especificación 2.1 del mixin](#).

Variables y métodos de clase

Utiliza la palabra clave `static` para implementar variables y métodos para toda la clase.

Variables estáticas

Las variables estáticas (variables de clase) son útiles para el estado y las constantes de toda la clase:

```

class Queue {
    static const initialCapacity = 16;
    // ...
}

void main() {
    assert(Queue.initialCapacity == 16);
}

```

Las variables estáticas no se inicializan hasta que se usan.

Nota: Esta página sigue la [recomendación de la guía de estilo de preferencia lowerCamelCase](#) para nombres constantes.

Métodos estáticos

Los métodos estáticos (métodos de clase) no funcionan en una instancia y, por lo tanto, no tienen acceso a `this`. Por ejemplo:

```
import 'dart:math';

class Point {
  num x, y;
  Point(this.x, this.y);

  static num distanceBetween(Point a, Point b) {
    var dx = a.x - b.x;
    var dy = a.y - b.y;
    return sqrt(dx * dx + dy * dy);
  }
}

void main() {
  var a = Point(2, 2);
  var b = Point(4, 4);
  var distance = Point.distanceBetween(a, b);
  assert(2.8 < distance && distance < 2.9);
  print(distance);
}
```

Nota: Considera utilizar funciones de alto nivel, en lugar de métodos estáticos, para utilidades y funcionalidades comunes o ampliamente utilizadas.

Puedes usar métodos estáticos como constantes en tiempo de compilación. Por ejemplo, puedes pasar un método estático como parámetro a un constructor de constantes.

Genéricos

Si observas la documentación de la API para el tipo de array básico, `List`, verás que el tipo es `List<E>`. Las marcas de notación `<...>` de `List`, son como un tipo *genérico* (o *parametrizado*), un tipo que tiene parámetros de tipo formal. Por convención, la mayoría de las variables de tipo tienen nombres de una sola letra, como E, T, S, K y V.

¿Por qué usar genéricos?

Los genéricos son a menudo requeridos para la seguridad del tipo, pero tienen más beneficios que simplemente permitir que su código se ejecute:

- Si se especifican correctamente los tipos genéricos, se obtiene un código mejor generado.
- Puedes usar genéricos para reducir la duplicación de código.

Si deseas que una lista contenga sólo strings, puedes declararla como `List<String>` (léase "lista de strings"). De esta manera, tanto tu, como tus compañeros programadores y tus herramientas pueden detectar que asignar un no-string a la lista es probablemente un error. Aquí hay un ejemplo:

```
var names = List<String>();
names.addAll(['Seth', 'Kathy', 'Lars']);
names.add(42); // Error
```

Otra razón para usar genéricos es reducir la duplicación de código. Los genéricos le permiten compartir una única interfaz e implementación entre muchos tipos, sin dejar de aprovechar el análisis estático. Por ejemplo, supongamos que crea una interfaz para almacenar en caché un objeto:

```
abstract class ObjectCache {
  Object getByKey(String key);
  void setByKey(String key, Object value);
}
```

Descubrirás que deseas una versión específica de string de esta interfase, por lo que crearás otra interfase:

```
abstract class StringCache {
  String getByKey(String key);
  void setByKey(String key, String value);
}
```

Más tarde, decides que quieres una versión específica de esta interfaz... Tienes la idea.

Los tipos genéricos pueden ahorrarte la molestia de crear todas estas interfaces. En su lugar, puedes crear una única interfase que tome un parámetro de tipo:

```
abstract class Cache<T> {
  T getByKey(String key);
  void setByKey(String key, T value);
}
```

En este código, T es el tipo de soporte. Es un marcador de posición que puede considerarse como un tipo lo que un desarrollador definirá más adelante.

Usando literales de la colección

Se pueden parametrizar los literales List, set y map. Los literales parametrizados son como los literales que ya has visto, excepto que añades `<type>` (para listas y conjuntos) o `<keyType, valueType>` (para mapas) antes de abrir el corchete. He aquí un ejemplo de uso de literales mecanografiados:

```
var names = <String>['Seth', 'Kathy', 'Lars'];
var uniqueNames = <String>{'Seth', 'Kathy', 'Lars'};
var pages = <String, String>{
  'index.html': 'Homepage',
  'robots.txt': 'Hints for web robots',
  'humans.txt': 'We are people, not machines'
};
```

Uso de tipos parametrizados con constructores

Para especificar uno o más tipos al usar un constructor, coloca los tipos entre corchetes angulares (`<...>`) justo después del nombre de la clase. Por ejemplo:

```
var nameSet = Set<String>.from(names);
```

El siguiente código crea un mapa que tiene claves enteras y valores de tipo Vista:

```
var views = Map<int, View>();
```

Colecciones genéricas y los tipos que contienen

Los tipos genéricos de Dart son *reificados* (Hacer algo abstracto más concreto o real), lo que significa que llevan su información de tipo en tiempo de ejecución. Por ejemplo, puedes probar el tipo de colección:

```
var names = List<String>();
names.addAll(['Seth', 'Kathy', 'Lars']);
print(names is List<String>); // true
```

Nota: En contraste, los genéricos en Java usan borradura, lo que significa que los parámetros de tipo genérico se eliminan en tiempo de ejecución. En Java, puedes probar si un objeto es una List, pero no puedes probar si es un `List<String>`.

Restricción del tipo parametrizado

Al implementar un tipo genérico, es posible que desees limitar los tipos de sus parámetros. Puedes hacerlo utilizando `extends`.

```
class Foo<T extends SomeBaseClass> {
  // Implementation goes here...
  String toString() => "Instance of 'Foo<$T>'";
}

class Extender extends SomeBaseClass {...}
```

Está bien usar `SomeBaseClass` o cualquiera de sus subclases como argumento genérico:

```
var someBaseClassFoo = Foo<SomeBaseClass>();
var extenderFoo = Foo<Extender>();
```

También está bien no especificar ningún argumento genérico:

```
var foo = Foo();
print(foo); // Instancia de 'Foo<SomeBaseClass>'
```

La especificación de cualquier tipo que no sea `SomeBaseClass` produce un error:

```
var foo = Foo<Object>(); //Error
```

Usando métodos genéricos

Inicialmente, el soporte genérico de Dart se limitaba a las clases. Una sintaxis más nueva, llamada *métodos genéricos*, permite escribir argumentos de tipo sobre métodos y funciones:

```
T first<T>(List<T> ts) {
  // Haz un trabajo inicial o control de errores, entonces...
  T tmp = ts[0];
  // Haz alguna comprobación o procesamiento adicional...
  return tmp;
}
```

Aquí el parámetro tipo genérico en `first (<T>)` te permite usar el argumento tipo `T` en varios lugares:

- En el tipo de retorno de la función (`T`).
- En el tipo de argumento (`List<T>`).
- En el tipo de una variable local (`T tmp`).

Para obtener más información sobre los genéricos, véase [Utilización de métodos genéricos](#).

Bibliotecas y visibilidad

Las directivas `import` y `library` pueden ayudarle a crear una base de código modular y compartible. Las bibliotecas no sólo proporcionan APIs, sino que son una unidad de privacidad: los identificadores que comienzan con un guión bajo (`_`) sólo son visibles dentro de la biblioteca. *Cada aplicación de Dart* es una biblioteca, incluso si no usa una directiva `library`.

Las bibliotecas se pueden distribuir utilizando paquetes. Véase [Pub Package y Asset Manager](#) para obtener información sobre pub, un gestor de paquetes incluido en el SDK.

Uso de bibliotecas

Se usa `import` para especificar cómo se utiliza un espacio de nombres de una biblioteca en el ámbito de otra biblioteca.

Por ejemplo, las aplicaciones web de Dart generalmente utilizan la biblioteca `dart:html`, las cuales se pueden importar de esta forma:

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

El único argumento necesario para `importar` es una URI que especifique la biblioteca. Para las bibliotecas incorporadas, la URI tiene el especial de `dart:` scheme. Para otras bibliotecas, puedes usar una ruta del filesystem o el `package:` scheme. El `package:` scheme especifica las librerías proporcionadas por un gestor de paquetes como la herramienta pub. Por ejemplo:

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

Nota: URI significa identificador uniforme de recursos. Las URL (localizadores de recursos uniformes) son un tipo común de URI. En contraste, los genéricos en Java usan borradura, lo que significa que los parámetros de tipo genérico se eliminan en tiempo de ejecución.

Especificando un prefijo de biblioteca

Si importas dos bibliotecas que tienen identificadores en conflicto, entonces puedes especificar un prefijo para una o ambas bibliotecas. Por ejemplo, si `library1` y `library2` tienen una clase `Element`, entonces podría tener un código como este:

```
import 'package:lib1/lib1.dart';
import 'package:lib2/lib2.dart' as lib2;

// Usa Element de lib1.
Element element1 = Element();
```



```
// Usa Element de lib2.
lib2.Element element2 = lib2.Element();
```

Importar sólo una parte de una biblioteca

Si deseas utilizar sólo una parte de una biblioteca, puedes importarla de forma selectiva. Por ejemplo:

```
// Importa únicamente foo.
import 'package:lib1/lib1.dart' show foo;

// Import todos los nombres EXCEPTO foo.
import 'package:lib2/lib2.dart' hide foo;
```

Cargando una biblioteca sin prisa (Lazily)

La *carga diferida* (*Deferred loading*), también llamada *carga lenta* (*lazy loading*) permite que una aplicación cargue una biblioteca bajo demanda, cuando y donde sea necesario. He aquí algunos casos en los que podrías utilizar la carga diferida:

- Para reducir el tiempo de arranque inicial de una aplicación.
- Para realizar pruebas A/B, por ejemplo, probando implementaciones alternativas de un algoritmo.
- Para cargar funcionalidades poco utilizadas, como pantallas y cuadros de diálogo opcionales.

Para cargar sin prisa una biblioteca, primero debes importarla usando `deferred as`.

```
import 'package:greetings/hello.dart' deferred as hello;
```

Cuando necesites la biblioteca, invoca `loadLibrary()` usando el identificador de la biblioteca.

```
Future greet() async {
  await hello.loadLibrary();
  hello.printGreeting();
}
```

En el código anterior, la palabra clave `await` detiene la ejecución hasta que se carga la librería. Para obtener más información sobre la `async` y `await`, consulta el [soporte de asincronía](#).

Puedes invocar `loadLibrary()` varias veces en una biblioteca sin problemas. La biblioteca se carga sólo una vez.

Cuando utilice la carga diferida, tenga en cuenta lo siguiente:

- Las constantes de una biblioteca diferida no son constantes en el archivo importado. Recuerde, estas constantes no existen hasta que se carga la librería diferida.
- No se pueden utilizar tipos de una biblioteca diferida en el archivo importado. En su lugar, considere mover los tipos de interfaz a una biblioteca importada tanto por la biblioteca diferida como por el archivo importado.
- Dart inserta implícitamente `loadLibrary()` en el espacio de nombres que se define utilizando `deferred as namespace`. La función `loadLibrary()` devuelve un `Future`.

Diferencia de Dart VM: Dart VM permite el acceso a miembros de bibliotecas diferidas incluso antes de la llamada a `loadLibrary()`. Este comportamiento puede cambiar, así que **no dependa del comportamiento actual de la máquina virtual**. Para más detalles, véase el [issue 33118](#).

Implementación de bibliotecas

Véase [Crear Paquetes de Biblioteca](#) para obtener consejos sobre cómo implementar un paquete de biblioteca, incluyendo:

- Cómo organizar el código fuente de la biblioteca.
- Cómo utilizar la directiva `export`.
- Cuándo usar la directiva `part`.
- Cuándo usar la directiva `library`.

Soporte de asincronía

Las bibliotecas de Dart están llenas de funciones que devuelven objetos `Future` o `Stream`. Estas funciones son *asíncronas*: regresan después de configurar una operación que puede llevar mucho tiempo (como la E/S), sin esperar a que la operación finalice.

Las palabras clave `async` y `await` son compatibles con la programación asíncrona, lo que permite escribir código asíncrono con un aspecto similar al del código síncrono.

Manejo de Futuros

Cuando necesite el resultado de un Futuro completo, tiene dos opciones:

- Usa `async` y `await`.
- Utilice la API Future, como se describe en la [visita guiada de la biblioteca](#).

El código que usa `async` y `await` es asíncrono, pero se parece mucho al código sincrónico. Por ejemplo, aquí hay un código que usa `await` para esperar el resultado de una función asíncrona:

```
await lookUpVersion();
```

Para usar `await`, el código debe estar en una función asíncrona, una función marcada como `async`:

```
Future checkVersion() async {  
  var version = await lookUpVersion();  
  // Hacer algo con la versión  
}
```

Nota: Aunque una función `async` puede realizar operaciones que consumen mucho tiempo, no espera a que se realicen. En su lugar, la función `async` se ejecuta sólo hasta que encuentra su primera expresión `await` (detalles). Luego devuelve un objeto Future, reanudando la ejecución sólo después de que se complete la expresión `await`.

```
try {  
  version = await lookUpVersion();  
} catch (e) {  
  // Reaccionar ante la incapacidad de buscar la versión.  
}
```

Puedes utilizar `await` varias veces en una función asíncrona. Por ejemplo, el siguiente código espera tres veces los resultados de las funciones:

```
var entrypoint = await findEntrypoint();  
var exitCode = await runExecutable(entrypoint, args);  
await flushThenExit(exitCode);
```

En `await expression`, el valor de `expression` es usualmente un Futuro; si no lo es, entonces el valor es automáticamente envuelto en un Futuro. Este objeto Futuro indica una promesa de devolver un objeto. El valor de `await expression` es lo que el objeto retornó. La expresión `await` hace que la ejecución se pause hasta que el objeto esté disponible.

Si se obtiene un error en tiempo de compilación al usar `await`, asegúrese de que `await` esté en una función `async`. Por ejemplo, para usar `await` en la función `main()` de su aplicación, el cuerpo de `main()` debe estar marcado como `async`:

```
Future main() async {
  checkVersion();
  print('In main: version is ${await lookUpVersion()}');
}
```

Declarar funciones asíncronas

Una *función asíncrona* es una función cuyo cuerpo está marcado con el modificador `async`.

Agregar la palabra clave `async` a una función hace que devuelva un Futuro. Por ejemplo, considere esta función síncrona, que devuelve un String:

```
String lookUpVersion() => '1.0.0';
```

Si se cambia para que sea una función asíncrona, por ejemplo, porque una implementación futura consumirá mucho tiempo, el valor devuelto es un Futuro:

```
Future<String> lookUpVersion() async => '1.0.0';
```

Ten en cuenta que el cuerpo de la función no necesita usar la API Future. Dart crea el objeto Future si es necesario.

Si su función no devuelve un valor útil, haga el tipo de retorno `Future<void>`.

Manejo de Streams

Cuando necesitas obtener valores de un Stream, tienes dos opciones:

- Utiliza `async` y un *for loop* asíncrono (`await for`).
- Utiliza la API de Stream, tal y como se describe en la [visita guiada de la biblioteca](#).

Note: Antes de usar `await for`, asegúrate de que el código sea más claro y de que realmente se desea esperar todos los resultados del stream. Por ejemplo, por lo general no debería utilizar `await for` para los escuchas de eventos de interfaz de usuario, ya que los frameworks de interfaz de usuario envían un sinfín de stream de eventos.

Un asíncrono para bucle tiene la siguiente forma:

```
await for (varOrType identifier in expression) {
  // Executes each time the stream emits a value.
}
```

El valor de `expression` debe tener el tipo Stream. La ejecución se lleva a cabo de la siguiente manera:

1. Espera hasta que el stream emita un valor.
2. Ejecutar el cuerpo del for loop, con la variable ajustada a ese valor emitido.
3. Repita los pasos 1 y 2 hasta que el stream se cierre.

Para detener la escucha del stream, puedes usar una sentencia `break` o `return`, la cual lo saca del bucle y cancela la suscripción al stream.

Si obtienes un error en tiempo de compilación al implementar un for loop asíncrono, asegúrate de que `await for` se encuentre en una función asíncrona. Por ejemplo, para usar un for loop asíncrono en la función `main()` de tu aplicación, el cuerpo de `main()` debe estar marcado como `async`:

```
Future main() async {
  // ...
  await for (var request in requestServer) {
    handleRequest(request);
  }
  // ...
}
```

Para obtener más información sobre la programación asíncrona, en general, consulta la sección [dart:async](#) de la visita guiada de la biblioteca. También puedes ver los artículos de [Dart Language Asynchrony Support: Fase 1](#) y [Soporte de Asincronía de Lenguaje de Dart: Fase 2](#), y la [especificación del lenguaje Dart](#).

Generadores

Cuando necesites producir lentamente una secuencia de valores, considera usar una *función generator*. Dart tiene soporte incorporado para dos tipos de funciones generator:

- Generador **síncrono**: Devuelve un objeto [Iterable](#).
- Generador **asíncrono**: Devuelve un objeto [Stream](#).

Para implementar una función de generador **síncrona**, marque el cuerpo de la función como `sync*` y utilice sentencias `yield` para entregar los valores:

```
Iterable<int> naturalsTo(int n) sync* {
  int k = 0;
  while (k < n) yield k++;
}
```

Para implementar una función de generador **asíncrona**, marque el cuerpo de la función como `async*` y utilice sentencias `yield` para entregar los valores:

```
Stream<int> asynchronousNaturalsTo(int n) async* {
  int k = 0;
  while (k < n) yield k++;
}
```

Si su generador es recursivo, puede mejorar su rendimiento utilizando el `yield*`:

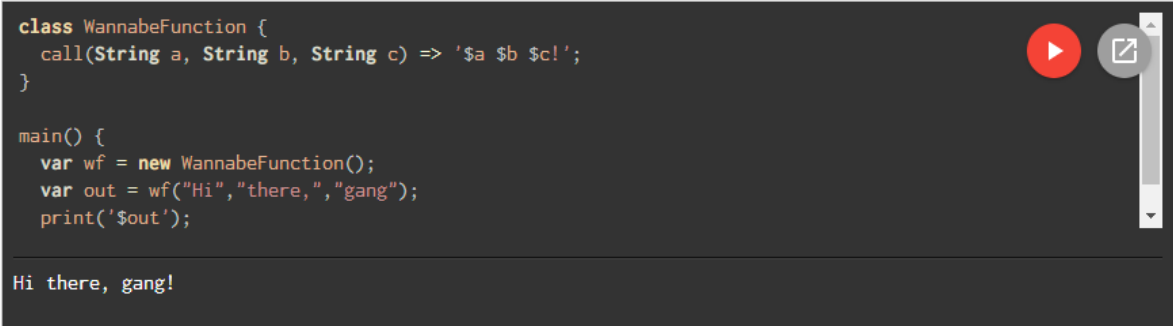
```
Iterable<int> naturalsDownFrom(int n) sync* {
  if (n > 0) {
    yield n;
    yield* naturalsDownFrom(n - 1);
  }
}
```

Para más información acerca de los generadores, véase el artículo [Soporte de Dart Language Asynchrony: Fase 2](#).

Clases invocables

Para permitir que su clase Dart sea llamada como una función, implementa el método `call()`.

En el siguiente ejemplo, la clase `WannabeFunction` define una función `call()` que toma tres strings y los concatena, separando cada uno con un espacio y añadiendo una exclamación.



```
class WannabeFunction {
  call(String a, String b, String c) => '$a $b $c!';
}

main() {
  var wf = new WannabeFunction();
  var out = wf("Hi", "there", "gang");
  print('$out');
}
```

Hi there, gang!

Para obtener más información sobre el tratamiento de clases como funciones, consulte [Emulación de funciones en Dart](#).

Isolates

La mayoría de los ordenadores, incluso en plataformas móviles, tienen CPUs multinúcleo. Para aprovechar todos estos núcleos, los desarrolladores utilizan tradicionalmente hilos de memoria compartida que se ejecutan simultáneamente. Sin embargo, la concurrencia de estados compartidos es propensa a errores y puede conducir a código complicado.

En lugar de hilos, todo el código de Dart se ejecuta dentro de los islotes. Cada isolate tiene su propio espacio de memoria, asegurando que el estado de ningún isolate sea accesible desde cualquier otro isolate.

Para obtener más información, consulta la [documentación de la biblioteca dart:isolate](#).

Typedefs

En Dart, las funciones son objetos, al igual que los strings y los números son objetos. Un *typedef*, o *alias de tipo función*, da a un tipo de función un nombre que puede utilizar al declarar campos y tipos de retorno. Un typedef retiene la información de tipo cuando se asigna un tipo de función a una variable.

Considere el siguiente código, que no utiliza un typedef:

```
class SortedCollection {
  Function compare;

  SortedCollection(int f(Object a, Object b)) {
    compare = f;
  }
}

// Implementación inicial, rota.
int sort(Object a, Object b) => 0;

void main() {
  SortedCollection coll = SortedCollection(sort);

  // Todo lo que sabemos es que compare es una función,
  // pero qué tipo de función?
  assert(coll.compare is Function);
}
```

La información de tipo se pierde cuando se asigna `f` a `compare`. El tipo de `f` es `(Object, Object) → int` (donde `→` significa returns), pero el tipo de `compare` es `Function`. Si cambiamos el código para usar nombres explícitos y retener información de tipo, tanto los desarrolladores como las herramientas pueden usar esa información.

```
typedef Compare = int Function(Object a, Object b);

class SortedCollection {
  Compare compare;

  SortedCollection(this.compare);
}

// Implementación inicial, rota.
```



```
int sort(Object a, Object b) => 0;

void main() {
  SortedCollection coll = SortedCollection(sort);
  assert(coll.compare is Function);
  assert(coll.compare is Compare);
}
```

Note: Actualmente, los typedefs están restringidos a tipos de funciones. Esperamos que esto cambie.

Metadatos

Utiliza metadatos para dar información adicional sobre tu código. Una anotación de metadatos comienza con el carácter `@`, seguido de una referencia a una constante en tiempo de compilación (como `deprecated`) o una llamada a un constructor de constantes.

Dos anotaciones están disponibles para todos los códigos de Dart: `@deprecated` y `@override`. Para ejemplos de uso de `@override`, véase [Extendiendo una clase](#). He aquí un ejemplo del uso de la anotación `@deprecated`:

```
class Television {
  /// _Deprecated: Use [turnOn] en lugar._
  @deprecated
  void activate() {
    turnOn();
  }

  /// Enciende el televisor.
  void turnOn() {...}
}
```

Puedes definir tus propias anotaciones de metadatos. Aquí hay un ejemplo de cómo definir una anotación de `@todo` que toma dos argumentos:

```
library todo;

class Todo {
  final String who;
  final String what;

  const Todo(this.who, this.what);
}
```

Y aquí hay un ejemplo del uso de esa anotación @todo:

```
import 'todo.dart';

@Todo('seth', 'make this do something')
void doSomething() {
  print('do something');
}
```

Los metadatos pueden aparecer ante una biblioteca, una clase, un typedef, un parámetro de tipo, un constructor, una fábrica, una función, un campo, un parámetro o una declaración de variables y ante una directiva de importación o exportación. Puedes recuperar metadatos en tiempo de ejecución utilizando la reflexión.

Comentarios

Dart soporta comentarios de una sola línea, comentarios de varias líneas y comentarios de documentación.

Comentarios de una sola línea

Un comentario de una sola línea comienza con `//`. Todo lo que se encuentre entre `//` y el final de la línea es ignorado por el compilador Dart.

```
void main() {
  // TODO: ¿refactorizar en un AbstractLlamaGreetingFactory?
  print('Welcome to my Llama farm!');
}
```

Comentarios de varias líneas

Un comentario de varias líneas comienza con `/*` y termina con `*/`. Todo lo que esté entre `/*` y `*/` es ignorado por el compilador de Dart (a menos que el comentario sea un comentario de documentación; véase la siguiente sección). Los comentarios de varias líneas pueden anidar.

```
void main() {
  /*
   * El suyo es mucho trabajo. Considera criar pollos.

   Llama larry = Llama();
   larry.feed();
   larry.exercise();
   larry.clean();
   */
}
```

Comentarios sobre la documentación

Los comentarios de documentación son comentarios de varias líneas o de una sola línea que comienzan con `///` o `/**`. El uso de `///` en líneas consecutivas tiene el mismo efecto que un comentario de documento multilínea.

Dentro de un comentario de documentación, el compilador de Dart ignora todo el texto a menos que esté entre corchetes. Utilizando paréntesis, puede hacer referencia a clases, métodos, campos, variables de alto nivel, funciones y parámetros. Los nombres entre paréntesis se resuelven en el ámbito léxico del elemento documentado del programa.

Aquí hay un ejemplo de comentarios de documentación con referencias a otras clases y argumentos:

```
/// Un camélido sudamericano domesticado (Lama glama).
///
/// Las culturas andinas han utilizado las llamas como carne y
/// animales de carga desde tiempos prehispánicos.
class Llama {
  String name;

  /// Alimenta a tu llama [Food].
  ///
  /// La llama típica come una bala de heno por semana.
```

```
void feed(Food food) {  
  // ...  
}  
  
/// Ejercita a tu llama con una[activity] por  
/// [timeLimit] en minutos.  
void exercise(Activity activity, int timeLimit) {  
  // ...  
}  
}
```

En la documentación generada, `[Food]` se convierte en un enlace a los documentos de la API para la clase `Food`.

Para analizar el código de Dart y generar documentación HTML, puedes utilizar la [herramienta de generación de documentación del SDK](#). Para ver un ejemplo de la documentación generada, consulta la [documentación de la API de Dart](#). Para consejos sobre cómo estructurar tus comentarios, consulta las [Pautas para los comentarios del Dart Doc](#).

Resumen

Esta página resume las características comúnmente usadas en el lenguaje de Dart. Se están implementando más características, pero esperamos que no rompan el código existente. Para más información, ve a la [especificación del lenguaje de Dart](#) y [Dart Efectivo](#).

Version 2.3.0, optimizada para la construcción de interfaces de usuario



Dart 2.3

El Release Estable del SDK de Dart 2.3, viene con nuevas características de lenguaje que mejoran su experiencia de codificación al desarrollar interfaces de usuario, nuevo soporte de herramientas para el desarrollo de la interfaz de usuario de Flutter, y dos nuevos sitios web: dart.dev y pub.dev. [Fuente]

Se agregan tres nuevas características para expresar la interfaz de usuario que está basada en listas, condicional o repetida.

Una interfaz es como un árbol de nodos de widgets. Algunos nodos contienen listas de widgets, por ejemplo, una lista de elementos deslizables. A menudo, estas listas se construyen a partir de otras listas. Para ello, se agregó una nueva característica al operador de extensión (spread) para desempaquetar los elementos de una lista en otra. En el siguiente ejemplo, `buildMainElements()` devuelve una lista de widgets, que luego se descomprime en la lista circundante utilizando el operador de extensión `...` :

```
Widget build(BuildContext context) {  
  return Column(children: [  
    Header(),  
    ...buildMainElements(),  
    Footer(),  
  ]);  
}
```

Código de ejemplo Dart utilizando el operador de propagación

Otra tarea común de la interfaz de usuario es incluir un elemento específico basado en una condición. Por ejemplo, es posible que desee incluir un botón *Next* en todas las páginas excepto en la última. Con Dart 2.3, puedes hacer esto usando una [colección if](#):

```
Widget build(BuildContext context) {
  return Column(children: [
    Text(mainText),
    if (page != pages.last)
      FlatButton(child: Text('Next')),
  ]);
}
```

Código de ejemplo Dart usando la colección if

Finalmente, las interfaces a menudo construyen elementos repetidos a partir de otros elementos repetidos. Puede expresarlo usando la nueva característica **colección for**:

```
Widget build(BuildContext context) {
  return Column(children: [
    Text(mainText),
    for (var section in sections)
      FlatButton(child: Text('Next')),
  ]);
}
```

Código de ejemplo Dart usando la colección for

También se han añadido nuevos *lints* que se pueden **configurar en el análisis estático** para que se apliquen utilizando las nuevas características de spread, la colección if y la colección for.

Nuevos sitios web de Dart & Pub

Se ha realizado la creación de un nuevo sitio web para la plataforma Dart: dart.dev

Este nuevo sitio presenta un landing page completamente nuevo, enfocado en explicar los beneficios principales de la plataforma Dart. También hemos actualizado las páginas de documentación para tener una mejor navegación y un mayor atractivo visual. Por último, hemos hecho una enorme reorganización de todo el contenido para facilitar el descubrimiento, y hemos añadido nuevas páginas para el contenido principal que antes faltaba.

También se ha actualizado visualmente el sitio del paquete Pub, pub.dev.

En Dev.XX.0

Cambios en la biblioteca del core

dart:async

Cambio de reparación: corrige un error en `StreamIterator` que permitía que el argumento del constructor fuera `null`. También permitía `await for` en un stream `null`. Esto es ahora un error de ejecución.

dart:core

Cambio de reparación: La interfaz de `RegExp` se ha ampliado con dos nuevos parámetros de construcción:

- `unicode`: (`bool`, default: `false`), para los patrones Unicode , y
- `dotAll`: (`bool`, default: `false`), para cambiar el comportamiento de coincidencia de '.' para que coincida también con los caracteres de terminación de línea.

También se han añadido las propiedades apropiadas para estos parámetros nombrados para que su uso pueda ser detectado después de la construcción.

Además, los métodos `RegExp` que originalmente devolvían objetos `Match` ahora devuelven un subtipo más específico, `RegExpMatch`, que añade dos características:

- `Iterable<String> groupNames`, una propiedad que contiene los nombres de todos los grupos de captura nombrados, y
- `String namedGroup(String name)`: un método que recupera la coincidencia para el grupo de captura dado.

Este cambio sólo afecta a los implementadores de la interfaz `RegExp`; el código actual que utiliza expresiones regulares de Dart no se verá afectado.

dart:isolate

Cambio de reparación: El `await for` permitía `null` como un stream debido a un error en la clase `StreamIterator` . Este error está corregido ahora.

Herramientas

Lint

Def: Análisis de código en busca de posibles errores

El Linter se actualizó a 0.1.88, que incluye los siguientes cambios:

- Solucionados los falsos positivos de `prefer_asserts_in_initializer_lists` (las listas de los preferidos en el inicializador)
- Solucionados las `curly_braces_in_flow_control_structures` (corchetes en estructuras de control de flujo) para manejar más casos
- Añadido nuevo lint: `prefer_double_quotes`
- Añadido nuevo lint: `sort_child_properties_last`
- Solucionados los falsos positivos `type_annotate_public_apis` para inicializadores `static const`

2.3.0

El enfoque en esta versión está en las nuevas características de lenguaje "UI-as-code" que hacen que las colecciones sean más expresivas y declarativas.

Lenguaje

Flutter está creciendo rápidamente, lo que significa que muchos usuarios de Dart están construyendo UI en código a partir de grandes expresiones profundamente anidadas. Nuestro objetivo con 2.3.0 consistía en **hacer que ese tipo de código fuera más fácil de escribir y mantener**. Los literales de las colecciones son un componente importante, por lo que nos focalizamos en tres características para que las colecciones sean más poderosas. Usaremos literales de lista en los ejemplos de abajo, pero estas características también funcionan en los literales de map y set.

Spread

Colocando `...` antes de una expresión dentro de una colección, el literal desempaqueta el resultado de la expresión e inserta sus elementos directamente dentro de la nueva colección. Donde antes tenías que escribir algo como esto:


```
CupertinoPageScaffold(
  child: ListView(children: [
    Tab2Header()
  ])..addAll(buildTab2Conversation())
    ..add(buildFooter()),
);
```

Ahora puedes escribir esto:

```
CupertinoPageScaffold(
  child: ListView(children: [
    Tab2Header(),
    ...buildTab2Conversation(),
    buildFooter()
  ]),
);
```

Si sabes que la expresión puede evaluar a nulo y quieres tratarla como equivalente a cero elementos, puedes usar el spread nulo-consciente `...?.`

Colección if

A veces es posible que desees incluir uno o más elementos en una colección sólo bajo ciertas condiciones. Si tienes suerte, puedes usar un operador `?:` para intercambiar selectivamente un solo elemento, pero si quieres intercambiar más de uno u omitir elementos, estás obligado a escribir un código imperativo como este:

```
Widget build(BuildContext context) {
  var children = [
    IconButton(icon: Icon(Icons.menu)),
    Expanded(child: title)
  ];

  if (isAndroid) {
    children.add(IconButton(icon: Icon(Icons.search)));
  }

  return Row(children: children);
}
```

Ahora permitimos `if` dentro de los literales de la colección para omitir condicionalmente o (con `else`) para intercambiar un elemento:

```
Widget build(BuildContext context) {
  return Row(
    children: [
      IconButton(icon: Icon(Icons.menu)),
      Expanded(child: title),
      if (isAndroid)
        IconButton(icon: Icon(Icons.search)),
    ],
  );
}
```

A diferencia del operador `?:` existente, una colección `if` puede componerse de extensiones para incluir u omitir condicionalmente múltiples ítems:

```
Widget build(BuildContext context) {
  return Row(
    children: [
      IconButton(icon: Icon(Icons.menu)),
      if (isAndroid) ...[
        Expanded(child: title),
        IconButton(icon: Icon(Icons.search)),
      ],
    ],
  );
}
```

Colección for

En muchos casos, los métodos de orden superior de Iterable ofrecen una forma declarativa de modificar una colección en el contexto de una única expresión. Pero algunas operaciones, especialmente las de transformación y filtrado, pueden ser engorrosas de expresar en un estilo funcional.

Para resolver este problema, puedes utilizar `for` dentro de una colección literal. Cada iteración del bucle produce un elemento que luego se inserta en la colección resultante. Considera el siguiente código:

```
var command = [
  engineDartPath,
  frontendServer,
  ...fileSystemRoots.map((root) => "--filesystem-root=$root"),
  ...entryPoints
    .where((entryPoint) => fileExists("lib/$entryPoint.json"))
    .map((entryPoint) => "lib/$entryPoint"),
  mainPath
];
```

Con una colección `for`, el código se vuelve más simple:

```
var command = [
  engineDartPath,
  frontendServer,
  for (var root in fileSystemRoots) "--filesystem-root=$root",
  for (var entryPoint in entryPoints)
    if (fileExists("lib/$entryPoint.json")) "lib/$entryPoint",
  mainPath
];
```

Como puede ver, estas tres características pueden ser compuestas libremente. Para más detalles sobre los cambios, véase [la propuesta oficial](#).

Nota: Estas características no están actualmente soportadas en los literales de la colección `const`. Un futuro lanzamiento, permitirá la difusión y la colección `if` dentro de las colecciones `const`.

Cambios en la biblioteca Core

`dart:isolate`

- Añadida la propiedad `debugName` a `Isolate`.
- Añadido el parámetro opcional `debugName` a `Isolate.spawn` y `Isolate.spawnUri`.

`dart:core`

- Los patrones `RegExp` ahora pueden usar afirmaciones de apariencia retrospectiva.
- Los patrones `RegExp` ahora pueden usar grupos de captura con nombre y referencias previas con nombre. Actualmente, las coincidencias de grupo

sólo pueden ser recuperadas en Dart ya sea por el índice implícito del grupo nombrado o por medio del descargue del objeto `Match` devuelto al tipo `RegExpMatch`. La interfaz `RegExpMatch` contiene métodos para recuperar los nombres de grupo disponibles y recuperar una coincidencia por nombre de grupo.

Dart VM

- El servicio VM ahora requiere un código de autenticación por defecto. Este comportamiento puede deshabilitarse proporcionando el indicador `--disable-service-auth-codes`.
- Se ha eliminado el soporte para las banderas obsoletas `-c` y `--checked`.

Dart para la web

dart2js

Se agregó un formato binario al `dump-info`. El antiguo formato JSON todavía está disponible y se proporciona por defecto, pero *estamos empezando a desaprobarlo*. El nuevo *formato binario* es más compacto y más barato de generar. En algunas aplicaciones grandes que probamos, era 4 veces más rápido de serializar y usaba 6 veces menos memoria.

Para usar el formato binario de hoy, usa `--dump-info = binary`, en lugar de `--dump-info`.

¿Qué esperar a continuación?

- La [herramienta del visualizador](#) no se actualizará para soportar el nuevo formato binario, pero puedes encontrar varias herramientas de línea de comandos en `package:dart2js_info` que proporcionan características similares a las del visualizador.
- Las herramientas de la línea de comandos del `package:dart2js_info` también funcionan con el antiguo formato JSON, por lo que puede empezar a utilizarlas incluso antes de activar el nuevo formato.
 - En una versión futura, `--dump-info` por defecto será `--dump-info=binario`. En ese momento, habrá una opción para volver al formato JSON, pero la herramienta del visualizador será obsoleta.
- Una versión posterior, el formato JSON ya no estará disponible en `dart2js`, pero puede estar disponible desde una herramienta de línea de comandos en el `package:dart2js_info`.

Herramientas

Dartfmt

- Modifique el formato literal para que siga el de otras colecciones.
- Añada soporte para las características "UI as code".
- Formatear correctamente las comas al final de las afirmaciones.
- Mejorar la sangría de las cadenas adyacentes en las listas de argumentos.

Linter

El Linter fue actualizado a 0.1.86, que incluye los siguientes cambios:

- Añadidas las siguientes pistas: `prefer_inline_adds`, `prefer_for_elements_to_map_fromIterable`, `prefer_if_elements_to_conditional_expressions`, `diagnostic_describe_all_properties`
- Se actualizaron los `file_names` para omitir los archivos Dart de extensión prefijada (`.css.dart`, `.g.dart`, etc.).
- Resueltos los falsos positivos en `unnecessary_parenthesis`.

Ciente pub

- Se agregó un validador CHANGELOG que se queja si publicas al public, `pub publish`, sin mencionar la versión actual.
- Se ha eliminado la validación de los nombres de las bibliotecas al hacer la `pub publish`.
- Añadido soporte para el paquete de activación `pub global activate` desde una URL pub personalizada.
- Añadido subcomando: `pub logout`. Salir de la sesión actual.

Dart nativo

- Se ha añadido soporte inicial para compilar aplicaciones Dart a código de máquina nativo. Se han añadido dos nuevas herramientas a la carpeta `bin` del SDK de Dart:
- `dart2aot`: AOT (ahead-of-time) compila un programa de Dart con código de máquina nativo. La herramienta es compatible con Windows, MacOS y Linux.
- `dartaotruntime`: Un pequeño tiempo de ejecución utilizado para ejecutar un programa compilado AOT.

Para obtener más información sobre las bibliotecas principales de Dart, ve [Un recorrido por las bibliotecas de Dart](#).

