Dart

Dart es un lenguaje de programación de código abierto desarrollado por Google. Dart es ideal para aplicaciones móviles y aplicaciones web. También puede ser usado en aplicaciones de escritorio, en aplicaciones de línea de comandos (*'command-line apps'*), como lenguaje para escribir scripts o en aplicaciones *server-side*.

Para aplicaciones web, **Dart Web** incluye un compilador en tiempo de desarrollo (*'dartdevc'*) y un compilador para producción (*'dart2js'*). La herramienta *'dart2js'* compila el código Dart en código JavaScript compacto, rápido y desplegable. Emplea técnicas como la eliminación de código muerto para generar código Javascript limpio y eficiente.

Para aplicaciones que se dirigen a dispositivos (móviles, de escritorio, servidores, etc..) **Dart Native** incluye una compilación Dart VM con compilación JIT (*'just-in-time'*) y un compilador AOT (*'ahead-of-time'*) para producir código máquina. **Dart Native** permite ejecutar código Dart compilado en código ARM nativo o X64 para aplicaciones móviles, de escritorio y de servidor.

Compilación AOT vs JIT

Históricamente, los lenguajes de programación se han dividido en dos grupos: **lenguajes estáticos** (por ejemplo, Fortran o C, donde las variables se escriben estáticamente en tiempo de compilación), y **lenguajes dinámicos** (por ejemplo, Smalltalk o JavaScript, donde el tipo de una variable puede cambiar en tiempo de ejecución). Los lenguajes estáticos se compilaban normalmente para producir programas de código nativo de máquina (o código ensamblador) para el equipo destino, que en tiempo de ejecución eran ejecutados directamente por el hardware. Los lenguajes dinámicos eran ejecutados por un intérprete, sin producir código de lenguaje de máquina.

Por supuesto, las cosas eventualmente se volvieron mucho más complicadas. El concepto de una máquina virtual (VM) se hizo popular, la cual no es más que un intérprete avanzado que imita a un hardware de máquina en software. Una máquina virtual facilita la transferencia de un lenguaje a nuevas plataformas de hardware. En este caso, el lenguaje de entrada de una máquina virtual suele ser un lenguaje intermedio. Por ejemplo, un lenguaje de programación (como Java) se compila en un lenguaje intermedio (*'bytecode'*) y luego se ejecuta en una VM (la JVM).

Además, ahora existen **compiladores JIT** (*'just-in-time'*). Un compilador JIT trabaja la ejecución del programa, compilando sobre la marcha. Los compiladores originales que se ejecutan durante la creación del programa (antes del tiempo de ejecución) se denominan ahora **compiladores AOT** (*'ahead-of-time'*).

En general, sólo los lenguajes estáticos son aptos para la compilación de AOT en código de máquina nativo porque los lenguajes de máquina normalmente necesitan saber el tipo de datos, y en los lenguajes dinámicos el tipo no se fija de antemano. En consecuencia, los lenguajes dinámicos suelen ser interpretados o compilados por JIT.

Cuando la compilación AOT se realiza durante el desarrollo, invariablemente resulta en ciclos de desarrollo mucho más lentos (el tiempo que transcurre entre el momento en que se realiza un cambio en un programa y el momento en que se puede ejecutar el programa para ver el resultado del cambio). Pero la compilación AOT da como resultado programas que pueden ejecutarse de forma más predecible y sin pausas para el análisis y la compilación en tiempo de ejecución. Los programas compilados por AOT también comienzan a ejecutarse más rápido (porque ya han sido compilados).

Por el contrario, la compilación JIT proporciona ciclos de desarrollo mucho más rápidos, pero puede dar lugar a una ejecución más lenta o jerárquica. En particular, los compiladores JIT tienen tiempos de inicio más lentos, porque cuando el programa comienza a ejecutarse, el compilador JIT tiene que hacer análisis y compilación antes de que el código pueda ser ejecutado.

Dart es uno de los pocos lenguajes que está bien adaptado para ser compilado tanto AOT como JIT. El soporte de ambos tipos de compilación proporciona ventajas significativas para Dart y (especialmente) Flutter.

La compilación JIT se utiliza durante el desarrollo, utilizando un compilador que es especialmente rápido lo que deriva en una de las características de Flutter, el *"stateful hot reload"*. Luego, cuando una aplicación está lista para su lanzamiento, se compila

AOT. Consecuentemente, con la ayuda de herramientas y compiladores avanzados, Dart puede ofrecer lo mejor de ambos mundos: ciclos de desarrollo extremadamente rápidos y tiempos de ejecución y puesta en marcha rápidos.

Dart puede ser compilado eficientemente AOT o JIT, interpretado o transpilado a otros lenguajes como Javascript.

Conceptos importantes

- Todo lo que puede colocar en una variable es un objeto, y cada objeto es una instancia de una clase. Incluso los números, las funciones y `null` son objetos. Todos los objetos heredan de la clase `Object`.
- Aunque Dart está fuertemente tipado, las anotaciones de tipo son opcionales porque Dart puede inferir tipos.
- Dart admite tipos genéricos, como `List<int>` (una lista de enteros) o `List<dynamic>` (una lista de objetos de cualquier tipo).
- Dart admite funciones de nivel superior o *'top-level functions'* (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).
- De manera similar, Dart admite variables de nivel superior, así como variables vinculadas a una clase u objeto (variables estáticas y de instancia). Las variables de instancia a veces se conocen 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.
- Los identificadores pueden comenzar con una letra o un guión bajo (_), seguido de cualquier combinación de esos caracteres más dígitos.
- Dart tiene expresiones (que tienen valores de tiempo de ejecución) y declaraciones (que no). Por ejemplo, la expresión condicional `condicional ? expr1 : expr2` tiene un valor u otro. Compare eso con una sentencia if-else, que no tiene valor. Una declaración a menudo contiene una o más expresiones, pero una expresión no puede contener directamente una declaración.
- Las herramientas de Dart pueden reportar dos tipos de problemas: **warnings** y **errors**. Las advertencias son solo indicaciones de que su código podría no funcionar, pero no impiden que su programa se ejecute. Los errores pueden ser de compilación o de ejecución. Un error en tiempo de compilación evita que el código se ejecute; un error en tiempo de ejecución hace que se genere una excepción mientras se ejecuta el código.

Compilar y ejecutar código Dart

```
// Define a function.
printInteger(int aNumber) {
  print('The number is $aNumber.'); // Print to console.
}

// This is where the app starts executing.
main() {
  var number = 42; // Declare and initialize a variable.
  printInteger(number); // Call a function.
}
```

Para ejecutar código en línea de comandos se necesita la VM de Dart que se incluye con el SDK de Dart. Una vez instalado, añadimos el *PATH* a las variables del sistema para ejecutar el comando `dart` directamente en la consola.

1. Crear un fichero llamado 'helloworld.dart':

```dart
void main() {
  print('Hello, World!');
}
```

1. Ejecutar el programa: `dart helloworld.dart`

Más información:

- https://dart.dev/tutorials/server/get-started
- https://dart.dev/tutorials/server/cmdline

Sintaxis básica

Comentarios

Dart admite comentarios de una sola línea (`//`), comentarios de varias líneas (`/* */`) y comentarios de documentación.

Los comentarios de una sola línea o de varias líneas funcionan **igual que en Java**.

```dart
void main() {
  // Comentarios de una sola línea
  print('Welcome to my Llama farm!');

  /*
  Comentarios de varias líneas
  larry.feed();
  larry.exercise();
  larry.clean();
  */
}
```

Los comentarios de documentación pueden ser de una línea o de varias líneas y empiezan por `///` o `/**`. Dentro de un comentario de documentación, el compilador ignora todo el texto a menos que esté entre corchetes. Usando corchetes, puede referirse a clases, métodos, campos, variables de nivel superior, funciones y parámetros. Los nombres entre paréntesis se resuelven en el ámbito léxico del elemento del programa documentado. Para generar la documentación, se usa la herramienta dartdoc

```dart
/// A domesticated South American camelid (Lama glama).
///
/// Andean cultures have used llamas as meat and pack
/// animals since pre-Hispanic times.
class Llama {
  /// Feeds your llama [Food].
  ///
  /// The typical llama eats one bale of hay per week.
  void feed(Food food) {
      // ...
  }

  /** Exercises your llama with an [activity] for
  [timeLimit] minutes. **/
  void exercise(Activity activity, int timeLimit) {
      // ...
  }
}
// En la documentación generada, [Food] se convierte en un enlace a los documentos API para la clase 'Food'.
```

Variables

Las variables almacenan referencias a objetos. La declaración de tipo es opcional ya que Dart puede inferir el tipo de variable. Dart tiene un **tipado fuerte** lo que significa que una variable de un tipo no puede almacenar referencias a objetos de otro tipo:

```
var name = 'John'; // El compilador infiere el tipo 'String'

String lastname = "Doe" // Declaración de tipo 'String'
// lastname = 40; // Error: A value of type 'dart.core::int' can't be assigned to a variable of type 'dart.core::String

var age = 40; // El compilador infiere el tipo 'int'
print(age.runtimeType); // => int
```

Si una variable no está restringida a un tipo, podemos indicar que es de tipo `Object` o `dynamic`. En este caso la variable puede almacenar distintos tipos:

```
dynamic name = 'Bob';
print(name.runtimeType); // => String

name = 45;
print(name.runtimeType); // => int
```

Las variables sin inicializar tienen un valor inicial `null`. Incluso las variables con tipos numéricos son inicialmente nulas, porque los números, como todo lo demás en Dart, son objetos.

`final` y `const`

Para declarar variables finales cuyo valor no va a cambiar, se utilizan las palabras clave `final` o `const` en lugar de la palabra clave `var`. Una variable `final` solo se puede asignar una vez; una variable `const` es una constante en tiempo de compilación. Una constante en tiempo de compilación o ***compile-time constant*** es una constante cuyo valor es conocido en tiempo de compilación.

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

```
final name = 'Bob'; // Without a type annotation
final String nickname = 'Bobby';

name = 'Alice'; // Error: a final variable can only be set once.
```

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

Utilice `const` para las variables que desea que sean constantes en tiempo de compilación (***compile-time constants***). Si la variable `const` está en el nivel de clase, márquela como `static const`.

Cuando se declare la variable como `const`, hay que establecer el valor en tiempo de compilación. Por tanto se necesita un número o cadena literal, otra variable `const` o el resultado de una operación aritmética en números constantes:

```
const bar = 1000000;
const double atm = 1.01325 * bar;
```

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 = []; // Equivalent to `const []`
```

Tipos incorporados

Dart tiene soporte especial para los siguientes tipos:

- numbers
- strings
- booleans
- lists (también conocido como arrays)
- maps
- runes
- symbols

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

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

Números

Dart tiene dos tipos para representar tipos numéricos. Ambos tipos son objetos.

- `int` - Valores enteros no mayores a **64 bits**, dependiendo de la plataforma. En Dart VM puede representar valores entre -2^63 y 2^63 - 1. Dado que Dart se puede compilar a Javascript, en Javascript el intervalo de valores está entre -2^53 hasta 2^53 - 1.

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

- `double` - Números de punto flotante de **64 bits** (precisión doble), según lo especificado por el estándar IEEE 754.

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

Tanto el tipo `int` como el tipo `double` son subtipos de la clase `num`. Esta clase incluye operaciones aritméticas como suma, resta, etc… y métodos como `abs()`, `ceil()`, `floor()`, etc…

Si la clase `num` y sus subtipos incluyendo las operaciones no son suficientes, la librería **dart:math** tiene una amplia variedad de tipos y métodos.

Se puede realizar la conversión entre tipos:

```
// 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');
```

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

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

Cadenas

Una cadena en Dart es una secuencia de unidades de código UTF-16. Para crear una cadena se pueden utilizar comillas simples o comillas dobles. El operador `+` permite concatenar cadenas:

```
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.";
var s5 = 'The + operator ' + 'works, as well.'; // => 'The operator + works, as well.'

var s6 = 'String '
    'concatenation'
    " works even over line breaks."; //=> 'String concatenation works even over line breaks'

var s7 = '''
You can create
multi-line strings like this one.
''';

var s8 = """This is also a
multi-line string.""";
```

En Dart (al igual que en Kotlin) se puede emplear expresiones y variables directamente en una cadena con la forma `${expresión}`. Si la expresión es un identificador, se pueden omitir las llaves `{}`. Esto se denomina **interpolación de cadena** o ***'template expressión'*** en Kotlin.

Para obtener la cadena correspondiente a un objeto, Dart llama al método `toString()` del objeto.

```
var s = 'string interpolation';

var s1 = 'Dart has $s.'; // => 'Dart has string interpolation.'
var s2 = 'Dart has ${s.toUpperCase()}.' // => 'Dart has STRING INTERPOLATION'
```

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

Las cadenas literales son constantes de tiempo de compilación, mientras que para que una interpolación de cadena sea una constante de tiempo de compilación las expresiones también tienen que ser constantes en tiempo de compilación.

```
// These work in a const string.
const aConstNum = 0;
const aConstBool = true;
const aConstString = 'a constant string';

// These do NOT work in a const string.
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';
```

Booleanos

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 de tiempo de compilación.

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 de tipo List.

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

Es importante destacar que en el ejemplo anterior el compilador infiere la lista como `List<int>`. Por tanto, cualquier intento de añadir objetos que no sean `int` el compilador lanzará un error.

Las listas, al igual que en la mayoría de lenguajes, utilizan la **indexación basada en cero**, donde 0 es el índice del primer elemento y `list.length - 1` es el índice del último elemento.

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

```
var constantList = const [1, 2, 3];
// constantList[1] = 1; // Uncommenting this causes an error.
```

Sets

En Dart, un conjunto es una colección desordenada de elementos únicos. El soporte de Dart para conjuntos se proporciona mediante literales de conjunto y el tipo Set.

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

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

```
var names = <String>{};
// Set<String> names = {}; // This works, too.
// var names = {}; // Creates a map, not a set.
```

Podemos añadir elementos a un conjunto existente con los métodos `add()` o `addAll()` y obtener el tamaño del conjunto con `.length`;

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

Para crear un conjunto que sea una constante en tiempo de compilación, hay que agregar `const` antes del literal del conjunto:

```
final constantSet = const {
  'fluorine',
  'chlorine',
  'bromine',
  'iodine',
  'astatine',
};
// constantSet.add('helium'); // Uncommenting this causes an error.
```

Maps

En general, un mapa es un objeto que asocia claves y valores. Tanto las claves como los valores pueden ser de cualquier tipo de objeto. Cada clave aparece solo una vez, pero puede usar el mismo valor varias veces. El soporte de Dart para mapas es proporcionado por literales de mapas y el tipo `Map`:

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

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

// Using constructor
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';
```

Destacar que, al igual que con los arrays, el compilador infiere los tipos. En el primer ejemplo, el compilador infiere el tipo como `Map<String, String>`. Añadir otro tipo al mapa genera un error.

```
var gifts = {'first': 'partridge'};
gifts['fourth'] = 'calling birds'; // Add a key-value pair

print(gifts['first']); // => partridge

print(gifts['fifth']); // => null
```

```
print(gifts.length); // => 2
```

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

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

// constantMap[2] = 'Helium'; // Uncommenting this causes an error.
```

Operadores

Cuando se usan los operadores, se crea una expresión.

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

Al igual que en otros lenguajes, hay operadores con mayor prioridad que otros. Para evitar problemas y facilitar la legibilidad, es recomendable usar paréntesis:

```
// Parentheses improve readability.
if ((n % i == 0) && (d % i == 0)) ...

// Harder to read, but equivalent.
if (n % i == 0 && d % i == 0) ...
```

Operadores aritméticos

```
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 (modulo)

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 sufijo.

++var	var = var + 1 (expression value is var + 1)
var++	var = var + 1 (expression value is var)
--var	var = var – 1 (expression value is var – 1)
var--	var = var – 1 (expression value is var)

```
var a, b;

a = 0;
b = ++a; // Increment a before b gets its value.
assert(a == b); // 1 == 1

a = 0;
b = a++; // Increment a AFTER b gets its value.
assert(a != b); // 1 != 0

a = 0;
b = --a; // Decrement a before b gets its value.
assert(a == b); // -1 == -1

a = 0;
b = a--; // Decrement a AFTER b gets its value.
assert(a != b); // -1 != 0
```

Igualdar y operadores relacionales

```
assert(2 == 2); // equal
assert(2 != 3); // not equal
assert(3 > 2);  // greater than
assert(2 < 3);  // less than
assert(3 >= 3); // greater than or equal to
assert(2 <= 3); // less than or equal to
```

Para probar si dos objetos representan la misma cosa, use el operador `==`. En el caso poco frecuente de necesitar saber si dos objetos son exactamente el mismo objeto, se usa la función `identical()`.

Operadores de comprobación de tipo

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

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

(emp as Person).firstName = 'Bob'; // Si 'emp' no es de tipo 'Person' se lanzará una excepción
```

Operadores de asignación

Para asignar un valor se usa el operador `=`. Para asignar un valor a una variable sólo si ésta es nula, se usa el operador `??=`.

```
// Assign value to a
a = value;
// Assign value to b if b is null; otherwise, b stays the same
b ??= value;
```

El símbolo y su expresión equivalente:

- `+=` : a = a + b
- `-=` : a = a - b

- `*=` : a = a * b
- `/=` : a = a / b
- `~/=` : a = a ~/ b
- `%=` : a = a % b
- `<<=` : a = a << b
- `=>>` : a = a >> b
- `&=` : a = a & b
- `^=` : a = a ^ b
- `|=` : a = a | b

```
var a = 2; // Assign using =
a *= 3; // Assign and multiply: a = a * 3
assert(a == 6);
```

Operadores lógicos

- `!exp` : inverts the following expression (changes false to true, and vice versa)
- `||` : logical OR
- `&&` : logical AND

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

Expresiones condicionales

Dart tiene dos operadores que permiten evaluar de forma concisa expresiones que de otro modo podrían requerir sentencias tipo `if-else` :

- `condición ? 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.

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

var b = a ?? 45; // si 'a' no es nulo se usa su valor o en caso contrario se usa 45 como valor para 'b'
```

La notación en cascada (..)

La notación en cascada (..) permite realizar una secuencia de operaciones en el mismo objeto. Además de las llamadas de función, también puede acceder a los campos en ese mismo objeto. Esto a menudo ahorra el paso de crear una variable temporal y permite escribir código más fluido y legible.

```
uerySelector('#confirm') // Get an object.
  ..text = 'Confirm' // Use its members.
  ..classes.add('important')
  ..onClick.listen((e) => window.alert('Confirmed!'));

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

La notación en cascada funciona cuando la función devuelve el objeto sobre el cual se aplica la siguiente función. Por ejemplo, la función `write()` de la clase `StringBuffer` no devuelve el objeto sino que devuelve `void`, con lo que el siguiente ejemplo produce un error:

```
var sb = StringBuffer();
sb.write('foo')
  ..write('bar'); // Error: method 'write' isn't defined for 'void'.
```

Control de flujo

If and else

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

Bucles for

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

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

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

Clases iterables como `List` y `Map` son compatibles con la forma `for-in`:

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

While and do-while

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

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

Break and continue

Se usa `break` para detener el bucle:

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

Se usa `continue` para saltarse la siguiente iteración del bucle:

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

Switch and case

En Dart los bloques `switch` comparan enteros, cadenas y constantes en tiempo de compilación usando `==`. Los objetos que se comparan tiene que ser de la misma clase. Las enumeraciones también funcionan con los bloques `switch`.

```
var command = 'OPEN';
switch (command) {
  case 'CLOSED': // Dart soporte las cláusulas vacías.
  case 'NOW_CLOSED':
    // Runs for both CLOSED and NOW_CLOSED.
    executeNowClosed();
    break;
  case 'PENDING':
    executePending();
    break;
  case 'APPROVED':
    executeApproved();
    break;
  case 'DENIED':
    executeDenied();
    break;
  case 'OPEN':
    executeOpen();
    break;
  default:
    executeUnknown(); // cláusula 'default'
}
```

Si se omite el `break` se produce un error:

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

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

Assert

Se usa `assert` para interrumpir la ejecución normal si una condición booleana es falsa. Cuando la condición es falsa la afirmación falla y se lanza la excepción `AssertionError`. Estas declaraciones son ignoradas en producción.

```
// Make sure the variable has a non-null value.
assert(text != null);

// Make sure the value is less than 100.
assert(number < 100);

// Make sure this is an https URL.
assert(urlString.startsWith('https'));

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

Funciones

Dart es un verdadero lenguaje orientado a objetos, por lo que incluso las funciones son objetos y tienen un tipo, el tipo Function. Esto significa que las funciones pueden asignarse a variables o pasarse como argumentos a otras funciones. También puede llamar a una instancia de una clase como si fuera una función.

```
bool isEven(int number) {
  return number % 2 == 0;
}

// Aunque se recomienda anotar el tipo en APIs públicas, se puede omitir
isEven(int number) {
  return number % 2 == 0;
}
```

Para las funciones que contienen una sola expresión, puede usar una sintaxis abreviada usando la notación `=>` también llamada *'arrow syntax'*. La sintaxis `=> expr` es una forma abreviada de `{ return expr; }`

```
bool isEven(int number) => number % 2 == 0;
```

Una función puede tener dos tipos de parámetros: **requeridos y opcionales**. Los parámetros requeridos se enumeran primero, seguidos de cualquier parámetro opcional. Los parámetros opcionales nombrados también se pueden marcar como `@required`.

Parámetros opcionales

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

Parámetros opcionales con nombre

Al definir una función que tenga parámetros con nombre se usa la forma `{param1, param2,…}` para especificar los parámetros nombrados:

```
// Named parameters
void enableFlags({bool bold, bool hidden}) {
  print("bold: $bold - hidden: $hidden");
}
```

Al llamar a una función, se especifican los parámetros con nombre usando `paramName: valor`:

```
enableFlags(); // => bold: null - hidden: null
enableFlags(bold: true); // => bold: true - hidden: null
enableFlags(bold: true, hidden: true); // => bold: true - hidden: true
enableFlags(hidden: true, bold: false); // bold: false - hidden: true
```

Para indicar que un parámetro es obligatorio usamos la anotación `@required` en el parámetro. Al usar la anotación el analizador de código del compilador pueden comprobar si no se suministra el argumento en la construcción y lanzar un aviso.

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

Parámetros opcionales por posición

Para indicar que uno o varios parámetros son posicionales se usan los corchetes `[]`:

```
String say(String from, String msg, [String device]) {
  print("from: $from - msg: $msg - device: $device");.
}

String hello(String from, [String device, bool status]) {
  print("from: $from - device: $device - status: $status");.
}

// Calling function
say('John', 'hi'); // => from: John - msg: hi - device: null
say('John', 'hi', 'phone'); // => from: John - msg: hi - device: phone

hello('John'); // => from: John - device: null - status: null
hello('John', 'phone'); // => from: John - device: phone - status: null
hello('John', 'phone', true); // => from: John - device: phone - status: true
```

Parámetros con valor por defecto

Para definir valores por defecto en parámetros con nombre y posicionales se usa `=`. Los valores por defecto deben ser constantes en tiempo de compilación. Si no se proporciona un valor por defecto, el valor predeterminado es `null`:

```
// Named parameters
void enableFlags({bool bold = false, bool hidden = false}) {
  //...
}

enableFlags(); // => bold: false - hidden: false
enableFlags(bold: true); // => bold: true - hidden: false
enableFlags(bold: true, hidden: true); // => bold: true - hidden: true
```

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

// Positional parameters
String say(String from, String msg, [String device = 'carrier pigeon']) {
  // ...
}

say('John', 'hi'); // => from: John - msg: hi - device: carrier pigeon
say('John', 'hi', 'phone'); // => from: John - msg: hi - device: phone
```

La función `main()`

Cada aplicación debe tener una función `main()` de nivel superior, que sirve como punto de entrada a la aplicación. La función `main()` devuelve `void` y tiene un parámetro opcional `List<String>` para los argumentos. Se puede usar la biblioteca 'args' para definir y analizar argumentos de línea de comandos.

```
// This is where the app starts executing.
main() {
  var number = 42; // Declare and initialize a variable.
  printInteger(number); // Call a function.
}

// Run the app like this: 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');
}
```

Functions as first-class objects

Las funciones se pueden pasar como parámetro a otra función:

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

var list = [1, 2, 3];

// Pass printElement as a parameter.
list.forEach(printElement);
```

También se puede asignar una función a una variable:

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

Funciones anónimas

Las funciones tienen un nombre que permite invocar la función pero también se pueden crear funciones sin nombre llamadas **funciones anónimas** o, a veces, *lambda* o *closure*. Se puede asignar una función anónima a una variable para, por ejemplo, agregarla o eliminarla de una colección.

Una función anónima es similar a una función con nombre. Pueden tener cero o más parámetros separados por comas y anotaciones de tipo opcionales entre paréntesis:

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

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

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

Si la función contiene solo una declaración, se puede acortar usando la notación de flecha `=>`. El código anterior puede escribirse como:

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

Ámbito léxico de las variables

Dart es un lenguaje de ámbito léxico, lo que significa que el alcance de las variables se determina de forma estática, simplemente por el diseño del código. Puede "seguir las llaves hacia afuera" para ver si una variable está dentro del alcance. Las funciones más interiores puede hacer uso de las variables de nivel superior.

```
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);
    }
  }
}
```

Lexical closures

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.

En el siguiente ejemplo, la función `makeAdder()` captura la variable `addBy`. Dondequiera que se utilice la función, 'recuerda' el valor de `addBy`.

```
/// Returns a function that adds [addBy] to the
/// function's argument.
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);
}
```

Testing functions for equality

Las *top-level functions*, los métodos estáticos y/o los métodos de instancia se pueden comparar entre sí:

```
void foo() {} // A top-level function

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

void main() {
  var x;

  // Comparing top-level functions.
  x = foo;
  assert(foo == x);

  // Comparing static methods.
  x = A.bar;
  assert(A.bar == x);

  // Comparing instance methods.
  var v = A(); // Instance #1 of A
  var w = A(); // Instance #2 of A
  var y = w;
  x = w.baz;

  // These closures refer to the same instance (#2),
  // so they're equal.
  assert(y.baz == x);

  // These closures refer to different instances,
  // so they're unequal.
  assert(v.baz != w.baz);
}
```

Valores de retorno

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

```
foo() {}

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

Excepciones

Las excepciones son errores que indican que sucedió algo inesperado. Si la excepción no se detecta, el *'isolate'* que generó la excepción se suspende y, por lo general, el *'isolate'* y su programa se terminan.

A diferencia de Java, todas las excepciones en Dart son *'unchecked exceptions'*. Los métodos no declaran las excepciones que pueden lanzar y no se requiere capturar ninguna excepción.

Dart provee los tipos `Exception` y `Error` así como otros subtipos. Además, se pueden definir excepciones propias o personalizadas.

Sin embargo, en Dart se puede lanzar cualquier objeto que no sea nulo como una excepción, no solo los objetos `Exception` y `Error`. No obstante, es recomendable utilizar estas clases, alguno de sus subtipos o nuestras propias excepciones.

```
throw FormatException('Expected at least 1 section'); // throws an exception

throw 'Out of llamas!'; // thow an arbitrary object
```

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

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

Al capturar una excepción se impide que la excepción se propague, a menos que se vuelva a lanzar la excepción. Capturar una excepción da la oportunidad de manejarla:

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

Para manejar código que puede lanzar más de un tipo de excepción, se puede especificar múltiples cláusulas de captura. La primera cláusula de captura que coincida 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 que sea lanzado:

```
try {
  breedMoreLlamas();
} on OutOfLlamasException {
  // A specific exception
  buyMoreLlamas();
} on Exception catch (e) {
  // Anything else that is an exception
  print('Unknown exception: $e');
} catch (e) {
  // No specified type, handles all
  print('Something really unknown: $e');
}
```

Como muestra el código anterior, se puede usar `on`, `catch` o ambos. Se utiliza `on` cuando se necesite especificar el tipo de excepción. Se utiliza `catch` cuando el manejador de excepciones necesite el objeto de excepción.

Se puede especificar uno o dos parámetros para `catch()`. El primer parámetro es la excepción que fue lanzada, y el segundo parámetro es la traza de la pila, que es 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 para luego ser relanzada para que se propague, se usa la palabra clave `rethrow`:

```
void misbehave() {
  try {
    dynamic foo = true;
    print(foo++); // Runtime error
  } catch (e) {
    print('misbehave() partially handled ${e.runtimeType}.');
    rethrow; // Allow callers to see the exception.
  }
}

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

Para asegurarse de que se ejecute código independientemente de si se lanza o no una excepción, se usa la cláusula `finally`. Si ninguna cláusula `catch` coincide con la excepción, la excepción se propaga después de que se ejecute la cláusula `finally`. Si una cláusula `catch` coincide y captura la excepción, se ejecuta esta cláusula y después se ejecuta la cláusula `finally`:

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

try {
  breedMoreLlamas();
} catch (e) {
  print('Error: $e'); // Handle the exception first.
} finally {
  cleanLlamaStalls(); // Then clean up.
}
```

Clases

Dart es un lenguaje orientado a objetos con clases y herencia de clases.

Miembros de clase

Los objetos tienen miembros que consisten en funciones y datos (métodos y variables de instancia, respectivamente). Cuando se llama a un método, se invoca en un objeto: el método tiene acceso a las funciones y datos de ese objeto. Se usa la notación punto `(.)` para referirse a una variable de instancia o un miembro de un objeto.

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

// Set the value of the instance variable y.
p.y = 3;

// Get the value of y.
assert(p.y == 3);

// Invoke distanceTo() on p.
num distance = p.distanceTo(Point(4, 4));
```

Se puede utilizar `?.` en lugar de solo `.` para evitar una excepción cuando el operando más a la izquierda sea nulo:

```
// If p is non-null, set its y value to 4.
p?.y = 4;
```

Crear una instancia

Puede crear un objeto utilizando un constructor. Los nombres de los constructores pueden ser *'ClassName'* o *'ClassName.identifier'*. La palabra clave `new` es opcional en Dart v2.0 y posteriores:

```
import 'dart:math';

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

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

Algunas clases proporcionan constructores constantes. Para crear una constante de tiempo de compilación utilizando un constructor constante, coloque la palabra clave `const` antes del nombre del constructor. La construcción de dos constantes idénticas en tiempo de compilación da como resultado una única instancia canónica:

```
import 'dart:math';

var p = const Point(2, 2);

var a = const Point(1, 1);
var b = const Point(1, 1);

assert(identical(a, b)); // They are the same instance!
```

Obtener el tipo de un objeto

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

```
var a = 45;
var b = Point(1, 1);
```

```
print('The type of a is ${a.runtimeType}'); // => The type of a is int
print('The type of a is ${b.runtimeType}'); // => The type of a is Point<int>
```

Variables de instancia

Las variables declaradas dentro de una clase son variables de instancia. Todas las variables de instancia sin inicializar tienen el valor nulo.

```
class Point {
  num x; // Declare instance variable x, initially null.
  num y; // Declare y, initially null.
  num z = 0; // Declare z, initially 0.
}
```

Todas las variables de instancia generan un método `getter` implícito. Las variables de instancia no final también generan un método `setter` implícito.

Si se inicializa 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 ejecute el constructor y su lista de inicializadores.

```
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.
}
```

Constructores

Un constructor se declara creando una función con el mismo nombre que la clase. La forma más común de constructor es el constructor que sirve para crear una nueva instancia de una clase. Dada su utilidad y lo común de su uso Dart tiene una forma compacta de constructor en la cual los argumentos del constructor se asignan a las variables de instancia:

```
class Point {
  num x, y;

  Point(num x, num y) {
    this.x = x; // 'this' se refiere a la instancia actual
    this.y = y;
  }
}

class Point {
  num x, y;

  // Forma compacta
  Point(this.x, this.y); // los argumentos 'x' e 'y' se asignan a las variables de instancia con el mismo nombre
}
```

Constructor por defecto

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

Las subclases no heredan constructores de su superclase. Una subclase que declara que no hay constructores tiene solo el constructor por defecto (sin argumento y sin nombre).

Constructores con nombre

Los constructores con nombre se emplean para implementar múltiples constructores para una clase o para proporcionar mayor claridad.

Como ya se ha comentado, los constructores no se heredan, lo que significa que el constructor con nombre de una superclase no es heredado por una subclase. En caso de que sea necesario que una subclase se cree con un constructor con nombre definido en la superclase, debe implementar ese constructor en la subclase.

```
class Point {
  num x, y;

  Point(this.x, this.y);

  // Named constructor
  Point.origin() {
    x = 0;
    y = 0;
  }
}
```

Lista de inicialización

Se puede inicializar las variables de instancia antes de que se ejecute el cuerpo del constructor. Los inicializadores se separan con comas:

```
class Point {
  var x, y;

  // Initializer list sets instance variables before
  // the constructor body runs.
  Point.fromJson(Map<String, num> json)
      : x = json['x'],
        y = json['y'] {
    print('In Point.fromJson(): ($x, $y)');
  }

  Point.raw(var x1, var y2) : x = x1, y = y2 { // initializer list
    print('In Point.raw(): ($x, $y)');
  }
}

void main() {
  Point point = Point.raw(10, 15); // => In Point.raw(): (10, 15)
}
```

Las listas de inicialización también son útiles al configurar variables finales:

```
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);
}
```

Durante el desarrollo, se puede validar entradas utilizando `assert` en la lista de inicializadores:

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

Invoking a non-default superclass constructor

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:

- lista de inicializadores
- el constructor de superclases sin argumentos
- constructor sin argumentos de la clase principal

Si la superclase no tiene un constructor sin nombre y sin argumentos, se debe llamar manualmente a uno de los constructores en la superclase. Para llamar a un constructor de la superclase se usa `:` justo antes del cuerpo del constructor si lo hay.

```
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({});

  // Prints:
  // in Person
  // in Employee
  if (emp is Person) {
    // Type check
    emp.firstName = 'Bob';
  }
  (emp as Person).firstName = 'Bob';
}
```

Debido a que los argumentos del constructor de la superclase se evalúan antes de invocar al constructor, un argumento puede ser una expresión como una llamada a la función. Sin embargo, los argumentos del constructor de la superclase no tienen acceso a `this`. Por ejemplo, los argumentos pueden llamar a métodos estáticos pero no a métodos de instancia.

```
class Employee extends Person {
  Employee() : super.fromJson(getDefaultData()); // 'getDefaultData() es un método estático
  // ···
}
```

Redirecting constructors

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

```
class Point {
  num x, y;

  // The main constructor for this class.
  Point(this.x, this.y);

  // Delegates to the main constructor.
  Point.alongXAxis(num x) : this(x, 0);
}
```

Constructores constantes

Si una clase instancia objetos que nunca cambian, se puede hacer que estos objetos sean constantes en tiempo de compilación. Para hacer esto, se define un constructor con la palabra clave `const` y todas las variables de instancia se definen como finales:

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

  final num x, y;

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

Factory constructors

Se usa la palabra clave `factory` cuando se implementa un constructor que no siempre crea una nueva instancia de su clase. Por ejemplo, un constructor puede devolver una instancia de una caché o puede devolver una instancia de un subtipo. Estos constructores se invocan como un constructor normal.

Métodos

Los métodos son funciones que proveen de comportamiento a los objetos.

Métodos de instancia

Los métodos de instancia pueden acceder a las variables de instancia y a `this`:

```
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. Cada variable de instancia tiene un *'getter'* implícito, más un *'setter'* si corresponde. Puede 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);

  // Define two calculated properties: right and 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);
}
```

Métodos abstractos

Los métodos de instancia, los *'getters'* y los *'setters'* pueden ser abstractos, definiendo una interfaz pero dejando su implementación en otras clases. Los métodos abstractos solo pueden declararse en **clases abstractas**.

```
abstract class Doer {
  // Define instance variables and methods...

  void doSomething(); // Define an abstract method.
}

class EffectiveDoer extends Doer {
  void doSomething() {
    // Provide an implementation, so the method is not abstract here...
  }
}
```

Clases abstractas

Para definir una clase abstracta se utiliza el modificador `abstract`. Una clase abstracta es una clase que no puede ser instanciada. Las clases abstractas son útiles para definir interfaces, a menudo con alguna implementación. Las clases abstractas normalmente tienen métodos abstractos.

```
// This class is declared abstract and thus
// can't be instantiated.
abstract class AbstractContainer {
  // Define constructors, fields, methods...

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

Interfaces implícitas

A diferencia de Java o Kotlin, en Dart no existe el concepto de interfaz como entidad. Cada clase define implícitamente una interfaz que contiene todos los miembros de instancia de la clase y de las interfaces que implementa. Si desea 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 al declararlas en una cláusula `implements` y luego proporcionar las API requeridas por las interfaces.

```
// A person. The implicit interface contains greet().
class Person {
  // In the interface, but visible only in this library.
  final _name;

  // Not in the interface, since this is a constructor.
  Person(this._name);

  // In the interface.
  String greet(String who) => 'Hello, $who. I am $_name.';
}

// An implementation of the Person interface.
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()));
}
```

Una clase puede implementar múltiples interfaces separadas por comas:

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

Extender una clase

Se utiliza la palabra clave `extends` para crear una subclase y la palabra clave `super` para referirse a la superclase:

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

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

Sobrescribir miembros

Las subclases puede sobrescribir métodos de instancia, *'getters'* y *'setters'*. Se utiliza la anotación `@override` para indicar al compilador que un método está sobrescribiendo un método de la superclase.

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

Tipos enumerados

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

Para declarar una enumeración se utiliza la palabra clave `enum`:

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

Cada valor en una enumeración tiene un índice, que devuelve la posición de dicho valor dentro de la enumeración, teniendo en cuenta que las enumeraciones, al igual que los arrays, empiezan en 0:

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

Para obtener una lista con todos los valores de una enumeración, se usa la constante `values`:

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

Las enumeraciones se puede emplear en bloques `switch`. En caso de que no haya una cláusula `case` para cada valor de la enumeración se lanza un aviso:

```
var aColor = Color.blue;

switch (aColor) {
  case Color.red:
    print('Red as roses!');
    break;
  case Color.green:
```

```
    print('Green as grass!');
    break;
  default: // Without this, you see a WARNING.
    print(aColor); // 'Color.blue'
}
```

Las clases enumeradas tienen los siguientes límites:

- No se puede heredar, mezclar o implementar un enumeración.
- No se puede instanciar explícitamente una enumeración.

Mixins

Más información

Class variables and methods

Se usa la palabra clave `static` para implementar variables y métodos en toda la clase.

Las variables estáticas (variables de clase) son útiles para constantes y estados de toda la clase. Las variables estáticas no se inicializan hasta que no se utilizan.

```
class Queue {
  static const initialCapacity = 16;
  // ···
}

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

Los métodos estáticos (métodos de clase) no funcionan en una instancia y, por lo tanto, no tienen acceso a `this`. Los métodos estáticos se pueden utilizar como constantes en tiempo de compilación.

```
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: Es recomendable utilizar funciones de nivel superior en lugar de métodos estáticos para utilidades y funcionalidades comunes que son ampliamente utilizadas.

Genéricos

Más información

Bibliotecas y visibilidad

Las directivas `import` y `library` permiten crear código modular y reutilizable. 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 en Dart es una biblioteca, incluso si no usa la directiva `library`. Las bibliotecas pueden distribuirse usando `packages`.

Utilizando bibliotecas

Se utiliza la palabra clave `import` para especificar cómo se usa un espacio de nombres de una biblioteca en el alcance de otra biblioteca, es decir, como importar un biblioteca para ser utilizada en otra biblioteca:

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

El único argumento necesario para importar una biblioteca es una URI que especifique la biblioteca. Para las bibliotecas incorporadas en el núcleo de Dart, la URI tiene la forma `dart:`. Para otras bibliotecas o bibliotecas de terceros, puede utilizar una ruta de sistema de archivos o la forma `package:`. La forma `package:` especifica las bibliotecas proporcionadas por un gestor de paquetes como la herramienta *'pub'*:

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

Especificar un prefijo

Si se importan dos bibliotecas que tienen identificadores en conflicto, se puede especificar un prefijo para una o ambas bibliotecas y así eliminar la ambigüedad:

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

// Uses Element from lib1.
Element element1 = Element();

// Uses Element from lib2.
lib2.Element element2 = lib2.Element();
```

Importar una porción de una biblioteca

Si desea utilizar solo una parte de una biblioteca, puede importar selectivamente la biblioteca:

```
// Import only foo.
import 'package:lib1/lib1.dart' show foo;

// Import all names EXCEPT foo.
import 'package:lib2/lib2.dart' hide foo;
```

Carga diferida de una biblioteca

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

- Para reducir el tiempo de inicio 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 indicar que una biblioteca se cargará de forma diferida se utiliza la palabra clave `deferred as`. Cuando la biblioteca sea necesaria, se invocará llamando al método `loadLibrary()`. En el ejemplo se emplea `await` para pausar la ejecución hasta que la biblioteca se cargue:

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

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

Tenga en cuenta lo siguiente cuando utilice la carga diferida:

- Puede invocar `loadLibrary()` varias veces en una biblioteca sin problemas. La biblioteca se cargará una sola vez.
- Las constantes de una biblioteca diferida no son constantes en el archivo de importación. Estas constantes no existen hasta que se carga la librería diferida.
- No puede utilizar tipos de una biblioteca diferida en el archivo de importación. En su lugar, considere mover los tipos de interfaz a una biblioteca importada tanto por la biblioteca diferida como por el archivo de importación.
- Dart inserta implícitamente `loadLibrary()` en el espacio de nombres cuando se utiliza `deferred as`. La función `loadLibrary()` devuelve un Future.

Asynchrony support

El conjunto de bibliotecas de Dart tienen muchas funciones que devuelven tipos como Future o Stream. Estas funciones son **asíncronas**: regresan después de configurar una operación que puede llevar mucho tiempo (como I/O), sin esperar a que esa operación se complete.

Dart proporciona las palabras clave `async` y `await` para dar soporte a la programación asíncrona, permitiendo escribir código asíncrono que se parece al código síncrono.

Existen dos formas de manejar el tipo *'Future'*:

- Uso de `async` y `await`
- Uso de la API de *'Future'* en la biblioteca `dart:async`

Una función asíncrona es una función cuyo cuerpo está marcado con el modificador `async`. Una función declarada como asíncrona retorna un tipo *'Future'*.

```
// Función síncrona
String lookUpVersion() => '1.0.0';

// Función asíncrona que retorna un Future<String>
Future<String> lookUpVersion() async => '1.0.0';

// Función asíncrona que retorna un Future<void> o Future
Future hello() async => print("Hello");
```

Aunque una función `async` puede realizar operaciones que requieren mucho tiempo, no espera a que se realicen. En su lugar, la función asíncrona se ejecuta sólo hasta que encuentra su primera expresión `await`. Luego devuelve un objeto *'Future'*, reanudando la ejecución sólo después de que se complete la expresión `await`.

Para manejar los errores se utiliza una sentencia `try-catch`:

```
try {
  version = await lookUpVersion();
} catch (e) {
  // React to inability to look up the version
}
```

Si se obtiene un error en tiempo de compilación al usar `await`, hay que comprobar que `await` esté en una función asíncrona. Es por esto que para usar `await` en la función `main()` esta se tiene que marcar como `async`:

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

Ejemplo de uso de `async` y `await`:

```
import 'dart:async';

Future<void> printDailyNewsDigest() async {
  var newsDigest = await gatherNewsReports();
  print(newsDigest);
}

main() {
  printDailyNewsDigest();
  printWinningLotteryNumbers();
  printWeatherForecast();
  printBaseballScore();
}

printWinningLotteryNumbers() {
  print('Winning lotto numbers: [23, 63, 87, 26, 2]');
}

printWeatherForecast() {
  print("Tomorrow's forecast: 70F, sunny.");
}

printBaseballScore() {
  print('Baseball score: Red Sox 10, Yankees 0');
}

const news = '<gathered news goes here>';
const oneSecond = Duration(seconds: 1);

// Imagine that this function is more complex and slow. :)
Future<String> gatherNewsReports() =>
    Future.delayed(oneSecond, () => news);
```

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 en Dart corre dentro de ***isolates***. Cada entorno aislado tiene su propia pila de memoria, lo que al no ser compartida se garantiza que no se pueda acceder ni modificar el estado.

Enlaces

Dart

- 🔶 Dart: Structured web apps
- 👓 https://github.com/yissachar/awesome-dart
- Find and use packages to build Dart and Flutter apps
  - Official packages published by the Dart Team
  - Firebase helps Flutter app teams succeed
- DartPad is an open-source tool that lets you play with the Dart language in any modern browser
- Jaspr | A modern web framework for building websites in Dart with support for both client-side and server-side rendering

Dart - Learning

- https://cheatsheets.zip/dart
- https://exercism.org/tracks/dart
- Dart Academy

Licencia