U1. Flutter.
El lenguaje de programación Dart

Contenidos

1 Introducción.

1.1 Sobre Flutter.

Flutter es un *framework* para el desarrollo de aplicaciones multiplataforma desarrollado por Google. Permite desarrollar aplicaciones compiladas nativas para cualquier plataforma a partir de una única base de de código. Está implementado con ***Dart***, y requiere el uso de *Dart* como lenguaje de programación.

Flutter está en pleno crecimiento. Si echamos un vistazo a los resultados del *Stack Overflow Survey* de 2023, vemos que *Flutter* está por encima del resto de frameworks para desarrollo de aplicaciones multiplataforma, seguido muy de cerca por *React Native*.[1]

El principal motivo del crecimiento de Flutter reside en que da lugar a aplicaciones multiplataforma que se ejecutan de forma nativa. El rendimiento de las aplicaciones es similar al de aquellas desarrolladas específicamente para cada plataforma. Flutter evita por completo los cuellos de botella y el impacto en el rendimiento que tienen otros frameworks basados en JavaScript, como Ionic.

Otro de los motivos, no menos importante, es que el aspecto de una aplicación Flutter es totalmente consistente de una plataforma a otra. No variará, ya que en lugar de usar las bibliotecas de componentes nativos de cada plataforma, Flutter ofrece una implementación propia para cada componente a nivel de pixel. Ello da lugar a un aspecto idéntico de una aplicación en cada una de las diferentes plataformas para las que se despliega.

Además, el proceso de desarrollo con Flutter es muy fluido e interactivo, ya que no es necesario construir y cargar la aplicación completa tras cada cambio. Eso significa que podemos ver el resultado de los cambios que hacemos en nuestro código, prácticamente de forma instantánea, en la aplicación en ejecución.

1.2 Sobre Dart.

Dart es el lenguaje de programación en que se basa Flutter, y en el que tenemos que trabajar si queremos desarrollar una aplicación con este framework. A continuación se muestra un listado con las características que lo hacen idóneo:

1. **La compilación y ejecución de Dart no solo es inusualmente flexible, sino que es especialmente rápida.**

   Dart es uno de los pocos lenguajes que se adapta bien para ser compilado tanto ***Ahead Of Time (AOT)*** como ***Just In Time (JIT)***. La compatibilidad de Dart con ambos tipos de compilación proporciona ventajas significativas para el desarrollo de aplicaciones con Flutter:

   a. La compilación JIT se usa durante el desarrollo, utilizando un compilador que es especialmente rápido. El código generado por el compilador JIT se ejecuta en la ***máquina virtual de Dart (Dart VM)***, la cual se incluye en los ejecutables que se generen, proporcionando acceso a la plataforma subyacente y permitiendo la recarga en caliente.

   b. Cuando una aplicación está lista para su lanzamiento, se compila AOT. Este compilador produce código nativo a la plataforma para la que se esté compilando, dando lugar a aplicaciones con un rendimiento excelente.

1 https://survey.stackoverflow.co/2023/#most-popular-technologies-misc-tech

Dart, por tanto, puede ofrecer lo mejor de ambos mundos: ciclos de desarrollo extremadamente rápidos y tiempos de ejecución y puesta en marcha rápidos.

2. **Dart se puede transpilar a código JavaScript.**

   Dart se puede transpilar a código *JavaScript* que puede ser ejecutado tanto en un navegador web como en un servidor con *Node.js*. Esto permite la reutilización de código entre aplicaciones móviles y aplicaciones web, y su uso en el lado del servidor, ya sea compilando en código nativo o compilando en *JavaScript* y usándolo con *Node.js*.

3. **Dart hace posible la recarga en caliente con estado (*hot reload*) de Flutter.**

   Gracias a esta característica, cualquier cambio que realicemos en el código fuente de nuestra aplicación se puede ver reflejado, prácticamente al instante, en la aplicación en ejecución. Con esto evitamos tener que recompilar y recargar al completo toda la aplicación, lo cual suele consumir bastante tiempo. Esta característica hace que sea mucho más fácil probar nuevas ideas o experimentar con alternativas, proporcionando un gran impulso a la creatividad.

   Para implementar esta característica, Flutter usa el compilador JIT de Dart para generar el fragmento de código afectado y seguidamente lo inyecta a la Dart VM en tiempo de ejecución, reemplazando al antiguo. La Dart VM intentará conservar el estado de la aplicación entre recargas siempre que sea posible.

4. **Dart permite implementar aplicaciones con un interfaz de usuario fluido y altamente interactivo.**

   El código Dart se ejecuta en un único hilo de ejecución, lo que denominamos un ***isolated*** (aislado). En este sentido, Dart funciona de modo similar a JavaScript: los aislados no comparten memoria, lo que evita el uso de bloqueos para acceder a zonas de memoria compartidas, y se comunican mediante paso de mensajes, de modo similar a como lo hacen los *workers* en JavaScript.

   El código en el propio *isolated* es el que determinará cuando inhibirá su ejecución, o cuando la retomará según las necesidades del momento, teniendo de este modo el desarrollador el control total en este aspecto. Por ejemplo:

   a. El código se inhibe a si mismo cada vez que efectúa una operación asíncrona (acceso a una API REST por HTTP o lectura de un fichero del sistema de archivos), y retomará la ejecución cuando la operación se haya resuelto y el resultado está disponible.

   b. La ejecución también puede retomarse debido a que se ha producido un evento que debe ser atendido.

   La programación asíncrona es, por tanto, muy utilizada en Dart, y esta forma de trabajar hace que la fluidez en las aplicaciones en ejecución sea excelente. Por tanto, tendremos interfaces de usuario más fluidas, a 120fps y con una respuesta instantánea, algo que sin duda es clave desde el punto de vista de la usabilidad de una aplicación.

5. **Dart es un excelente lenguaje de programación, ya que es moderno, robusto, *open source*, y está fuertemente reespaldado por Google.** Sus principales características como lenguaje son las siguientes:

   a. Es orientado a objetos puro. Esto significa que todo en él son objetos, incluidos los tipos de datos primitivos, como números o booleanos. Además, cuenta con un eficiente recolector de basura, e que implementa herencia simple con interfaces.

b. Su sintaxis es similar a la de otros lenguajes de programación, lo cual facilita su aprendizaje. A su vez, posee ciertas innovaciones sintácticas que permiten su uso para la definición completa de interfaces de usuario, sin necesidad de tener que usar un lenguaje de marcas tipo XML o HTML.

c. Soporta tipado de datos tanto estático como dinámico, aunque este último no es recomendable. Sus sistema de tipos es muy robusto y ofrece seguridad frente a valores nulos, lo que hace posible que nuestro IDE detecte una gran cantidad de errores en tiempo de desarrollo. Incluye soporte para tipos genéricos.

d. Buen soporte para la programación funcional y asíncrona, con características como las funciones arrow y los `Futures` (similares a los `Promises` de JavaScript).

e. Proporciona una completa colección de bibliotecas estándares que cubren un amplio abanico de necesidades.

1.3 El proceso de ejecución de código Dart.

El código Dart es ejecutado, al igual que sucede en otros lenguajes de programación, por una máquina virtual (la ***Dart VM***); sin embargo, la forma de hacerlo varía según las necesidades. Tenemos las siguientes opciones a la hora de utilizar el compilador:

1. ***Ejecución nativa***. El compilador generará código apto para ser ejecutado en una plataforma determinada (Windows 10, Linux, Android, IOS, etc.). Ten en cuenta que no es posible la compilación cruzada entre sistemas *Microsoft-Apple*. Dos modos de trabajo: depuración y producción.

    a. **Depuración**. Se genera un ejecutable que incluye la Dart VM, los servicios de depuración y los servicios de recolección de métricas. Se aplica a nuestro código un proceso de compilación JIT, que combinado con el mecanismo de hot reload, permite ver los cambios realizados en el código fuente de forma prácticamente instantánea en la aplicación en ejecución.

    b. **Producción**. El código de nuestro proyecto se compila junto al de la propia Dart VM en un proceso de compilación AOT, dando lugar a código nativo específicamente preparado para la plataforma en que se va a ejecutar el programa.

2. ***Ejecución en el navegador***. Nuestro programa se transpila a código JavaScript que se puede ejecutar en un navegador web. También tenemos los modos de desarrollo y producción.

1.4 Configuración del entorno de desarrollo.

En este apartado vas a instalar y configurar todo lo necesario para poder empezar a desarrollar y depurar aplicaciones Android con Flutter. Realiza los siguientes pasos:

1. Instala y configura ***Flutter***.

    a. Descargamos el `zip` de la última versión de *Flutter* desde su sitio web oficial.

    b. Lo descomprimimos donde queramos en nuestro sistema de archivos (por ejemplo, dentro del directorio personal de nuestro usuario).

2. Instala ***Android Studio***, instala las ***Android SDK Command Line Tools*** *y crea un* ***Emulador***.

    a. Descarga *Android Studio* de su sitio web oficial e instálalo.

b. Abrimos el *sdkmanager* e instalamos ***Android SDK Command Line Tools (latest)***. Está en la pestaña *SDK Tools*.

c. La instalación de *Android Studio* por defecto creará un emulador para un dispositivo *Pixel 3a* con la API 34, que es bastante reciente.

   Además de este, vas a crear otro para un dispositivo Pixel 3a con la API 26, lo que te permitirá probar tus futuras aplicaciones en una API más antigua, pero a su vez más extendida. Para ello tendrás que descargar e instalar tanto el SDK 26, como la imagen del emulador para esta API.

   ¿Por qué la API 26? Ver https://www.megumethod.com/blog/recommended-minimum-sdk-version-for-android-projects para más detalles.

3. Añade los siguientes directorios a la ***variable de entorno PATH***:

   1. `%PROGRAMFILES%\flutter\bin`
   2. `%HOMEPATH%\AppData\Local\Android\Sdk\platform-tools`
   3. `%HOMEPATH%\AppData\Local\Android\Sdk\cmdline-tools\latest\bin`

   Observa que `%HOMEPATH%` es la ruta del directorio personal de tu usuario en *Windows*, mientras que `%PROGRAMFILES%` es la ruta del directorio *Archivos de programa*.

4. Usa ***flutter doctor*** para verificar que el entorno está correctamente configurado.

   a. Asegurate de que el servicio “Background Intelligent Transfer Service” de Windows está activado. Si no es así, activalo y configúralo para que se autoejecute al arrancar el sistema.

   b. Ejecuta el comando `flutter doctor` desde la `CMD`.

   c. Si te avisa de que no has aceptado la licencia de uso de algún SDK, introduce el siguiente comando para aceptarlas: `flutter doctor –android-licenses`

   d. Vuelve a ejecutar `flutter doctor` para ver que todos los puntos de verificación son correctos.

5. Instala ***Visual Studio Code***.

   a. Descárgalo de su sitio web oficial e instálalo.

   b. Abre *VSCode* e instala la extensión oficial de *Flutter*, que también incluirá la extensión de *Dart*.

1.5 Tu primer proyecto con Dart: “Hola mundo”.

1.5.1 Creación de un nuevo proyecto.

Lo primero es crear un nuevo proyecto, al que llamaremos `holamundo`. Para ello tienes dos opciones: hacerlo desde la línea de comandos, o desde VSCode. A continuación se describen las dos opciones para que escojas la que prefieras.

a) Desde la consola.

Si hemos configurado correctamente nuestro entorno de desarrollo, el comando `dart` debería estar disponible en la línea de comandos. Procederemos a crear un nuevo proyecto del siguiente modo:

1. Abrimos una línea de comandos CMD.
2. Ejecutamos el siguiente comando para crear un proyecto *Dart* denominado `helloworld`.

```
dart create -t console-full helloworld
```

Tras la ejecución de este comando se creará un nuevo proyecto `helloworld` dentro de un directorio con igual nombre. Este directorio contendrá el esqueleto básico del proyecto, con los recursos mínimos necesarios para que podamos ejecutar un proyecto en modo consola.

3. Ejecuta el proyecto accediendo a su directorio correspondiente e introduciendo el siguiente comando:

```
dart run
```

b) Desde VSCode.

Podemos crear un nuevo proyecto usando el comando Dart: New project, disponible a través de la paleta paleta de comandos de VSCode. Para ello:

1. Presiona *Ctrl+Shift+P* y escribe `Dart: New project`.
2. Escoge el tipo de proyecto que deseas crear. En este caso será un proyecto en modo consola.
3. Selecciona el directorio donde se creará el nuevo proyecto.
4. Escribe un nombre para tu proyecto (`holamundo`) y presiona *ENTER*.

Tras esto se creará un nuevo proyecto en la ubicación seleccionada, y se abrirá el directorio de ese nuevo proyecto en una instancia de VSCode.

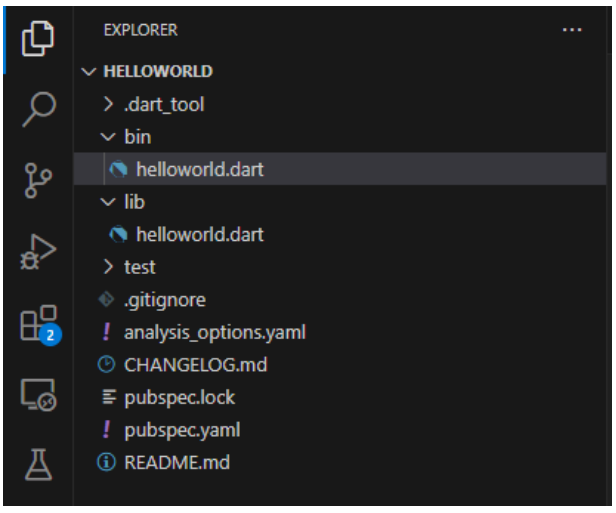
1.5.2 Uso de VSCode para trabajar sobre el proyecto.

a) Apertura del proyecto.

Si has optado por crear el proyecto desde la línea de comandos, lo primero que debes hacer es abrir VSCode y establecer como directorio de trabajo el directorio del nuevo proyecto:

1. Abre VSCode y asegúrate de tener instalada las extensión oficial de Dart (debería estarlo desde el paso 3 del punto 1.2).
2. Vé al menú *Archivo* y elige la opción *Abrir directorio*. También puedes presionar las teclas *Ctrl+K*, y seguidamente *Ctrl+O*.
3. Busca el directorio del proyecto (`holamundo`) y haz clic en “Seleccionar carpeta”.

   Tras esto, VSCode mostrará el contenido del directorio `holamundo` en el *Explorador*, tal y como se ve en la figura de la derecha.



b) Navegación y edición de los archivos del proyecto.

Una vez abierto, podemos hacer doble clic sobre cualquiera de los archivos que muestra el explorador para ver su contenido en el editor. Nos interesan los archivos `bin/helloworld.dart` y `lib/helloworld.dart`.

El contenido de `helloworld.dart` es el siguiente:

```dart
import 'package:helloworld/helloworld.dart' as helloworld;

void main(List<String> arguments) {
  print('Hello world: ${helloworld.calculate()}!');
}
```

Como puedes observar, en primer lugar se importa el código del archivo `helloworld.dart` del directorio `lib` a modo de paquete, y se le asigna el nombre local `helloworld`.

A continuación encontramos la función `main()`. Esta función es el punto de entrada de nuestro programa y será invocada automáticamente cuando lo ejecutemos. Es similar a lo que sucede en otros lenguajes de programación como C++ o Java.

Con respecto a su contenido, vemos que lo único que hace es usar la función `print()` para escribir un mensaje por la consola. El mensaje en cuestión es la cadena de texto `'Hello world:'` seguida del valor devuelto por la función `calculate()`, implementada dentro del archivo `/lib/helloworld.dart`.

Para ver el código de esta función, podemos abrir el archivo `/lib/helloword.dart`. Su contenido es el siguiente:

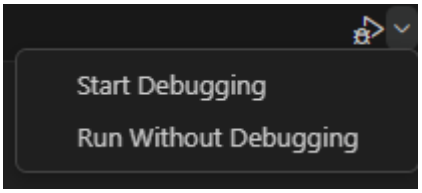
```dart
int calculate() {
  return 6 * 7;
}
```

Como podemos observar, la función `calculate()` no realiza nada particularmente interesante, simplemente devuelve el resultado de multiplicar `6*7`.

c) Ejecución del proyecto.

Podemos ejecutar fácilmente el proyecto. Para ello tenemos diferentes opciones:

- Accede al menú Ejecutar de VSCode y elige una de las siguientes opciones:
  - Iniciar depuración (o pulsar *F5*). Esta es la opción más común mientras estamos desarrollando nuestro proyecto. Al hacer uso de esta opción, veremos la salida del proyecto en la consola de depuración de VSCode.
  - Ejecutar sin depurar: *Ctrl+F5*. Ejecuta el proyecto sin la opción de depuración.
- Sitúate en el archivo que contiene la función `main()` (en este caso sería `bin/helloworld.dart`) y haz clic en el botón situado en la esquina superior derecha. La opción predeterminada será la de iniciar la depuración.

  Si nuestro objetivo no es depurar el proyecto, podemos hacer clic en la flechita a la derecha del botón y escoger la opción *"Run Without Debugging"*.

1.6 Escritura y lectura de datos por la consola.

1.6.1 Escritura de datos por la consola.

Para escribir datos en la consola podemos usar la función `print()`. Esta función recibe un objeto y escribe el resultado de su conversión a cadena de caracteres por la consola.

```
void main() {
  String nombre = "Juan";
  int edad = 30;

  // Usando print para mostrar información con saltos de línea automáticos
  print("Hola, soy $nombre.");
  print("Tengo $edad años.");
}
```

El método `stdout.write()`, muy similar a `print()`, ofrece una manera alternativa de escribir por la consola. Su funcionamiento es similar a `print()`, pero con la diferencia de que no escribirá un salto de línea al final. En caso de que deseemos un salto de línea, basta con añadir un `\n`, o bien usar el método `stdout.writeln()`.

```
import 'dart:io';

void main() {
  String nombre = "Ana";
  int edad = 25;

  // Las siguientes llamadas escribirán el texto en la misma línea, ya que el método write no
  // añade saltos de línea al final de forma automática
  stdout.write("Hola, soy $nombre. ");
  stdout.write("Tengo $edad años.");
}
```

Observa que para usar `stdout.write()` es necesario importar la librería `io`.

1.6.2 Lectura de datos desde la consola.

Para la lectura de datos desde la consola podemos usar el método `stdin.readLineSync()`. Ejemplo:

```
import 'dart:io';

void main() {
  stdout.write("¿Cómo te llamas? ");
  String nombre = stdin.readLineSync();
  print("Hola, $nombre!");
}
```

2 Variables y la sentencia de asignación.

A partir de este punto vamos a explorar el uso y características de Dart como lenguaje de programación, sin entrar aún a trabajar con Flutter.

2.1 Variables nullables, no nullables y su declaración.

En Dart, las variables se declaran de forma similar a como se hace en C#, Java o C++, salvo por una cuestión: en Dart, las variables pueden ser nullables o no nullables. Una ***variable nullable*** es aquella a la que se le puede asignar `null`. Una ***variable no nullable*** es aquella a la que no se le puede asignar `null`.

Nota: Los ejemplos que se muestran en este apartado hacen uso de los tipos de datos `int` y `double`, que se usan para declarar variables que permiten almacenar números enteros y decimales, respectivamente; pero hay muchos más tipos de datos disponibles, e incluso podemos definir los nuestros. Los veremos en el apartado 2.2.

2.1.1 Variables no nullables.

Para declarar una variable no nullable indicamos el tipo de dato seguido del nombre de la variable, y seguidamente le asignamos su valor inicial:

```
int numero = 5;
```

Esta declaración define una nueva variable numero, de tipo *entero no nullable*. Esto significa que numero no puede contener `null`. Aunque no pueda ser `null`, no es obligatorio asignarle un valor inicial en la propia declaración de la variable. Lo siguiente es perfectamente válido:

```
int numero;
numero = 5;
```

Lo que no se permite, y generará error de compilación, es hacer uso de la variable no nullable sin antes haberle asignado un valor. Por ejemplo, lo siguiente generaría un error de compilación:

```
int numero;
print(numero);     // Error: no se puede usar la variable numero sin antes haberla inicializado
```

2.1.2 Variables nullables.

Si queremos que una variable admita el valor `null`, debemos añadir el símbolo `?` al final del tipo de la variable. En este caso deja de ser obligatorio inicializar la variable en la propia declaración:

```
int? numero;       // No hay ningún problema, ya que numero es un entero nullable
print(numero);     // No hay ningún problema, mostraría null por la consola
numero = 5;
```

2.1.3 Declaración múltiple de variables.

Podemos declarar más de una variable en una misma sentencia:

```
int a = 2, b = -3;
```

2.1.4 Inferencia de tipo.

El compilador de Dart puede inferir el tipo de una variable de forma automática a partir del valor con que la inicializamos. Si queremos usar esta característica, podemos declarar las variables mediante la palabra reservada `var`, en lugar de indicar explícitamente su tipo de dato. Ejemplo:

```
var n = 5; // La variable n será de tipo int, inferido a partir del valor que se le asigna (5)
```

Es importante tener en cuenta que la inferencia de tipos se produce en tiempo de compilación. Por ello, una vez le es asignado el valor a una variable declarada con `var`, su tipo quedará fijado y no podrá cambiar.

2.1.5 Declaración de variables con tipado dinámico.

En Dart también podemos usar variables con tipado dinámico; es decir, variables cuyo tipo solo puede conocerse en tiempo de ejecución, y que puede variar dinámicamente. Ejemplo:

```
dynamic v;  // Declaramos una variable v con tipo dinámico
v = 1;      // El tipo de la variable cambia a int, y se le asigna el valor 1
print(v);
v = 'Hola'; // El tipo de la variable cambia a String, y se le asigna 'Hola'
print(v);
```

Este tipo de variables funciona de manera similar a como lo hacen las de JavaScript, y resultan útiles para diversas tareas, como por ejemplo, la manipulación de datos en formato *JSON*. Sin embargo, su uso debe reservarse solo para circunstancias especiales, siendo recomendable tipar nuestras variables estáticamente siempre que se pueda.

2.2 La sentencia de asignación.

Podemos modificar el valor de una variable en cualquier punto de nuestro programa mediante el uso de la ***sentencia de asignación***. Para ello, usaremos el operador `=` tal y como se muestra a continuación:

```
int? a;
a = 3;            // Asignamos a la variable a el valor 3
```

Como es lógico, al realizar asignaciones los tipos de datos de los valores que se asignan y las variables a las que son asignados deben ser compatibles. Por ello, es evidente que no podemos asignar un número con decimales a una variable de tipo `int`. Lo siguiente generaría un error de compilación:

```
double b = 3.14;
a = b;            // Genera un error de compilación, ya que a es de tipo int y b es double
```

2.2.1 Asignaciones y nulabilidad.

En este aspecto hay que tener en cuenta también si los tipos son compatibles en cuanto a valores nulos. Por ejemplo, a una variable de tipo `int?` le podemos asignar el valor de una variable de tipo `int` sin ningún problema. Sin embargo, lo contrario generará un error de compilación. Ejemplo:

```
int? a = 5;
int b = 3;
a = b;      // No hay ningún problema
```

```
b = a;      // Genera un error de compilación, ya que a podría valer null, no admitido por b
```

2.2.2 Conversión de una variable nullable en no nullable con el operador !

Para poder realizar la asignación `b = a` anterior sin que se genere un error de compilación debemos usar el operador !, tal y como se indica a continuación:

```
b = a!;           // No generará error de compilación, ya que ! transforma a en una variable
                  // no nullable mediante una conversión de tipo
```

No obstante, hay que tener en cuenta que si `a` es `null`, la asignación anterior generará una excepción en tiempo de ejecución. Solo debemos hacer uso de ! si estamos seguros en ese punto del programa de que `a` no contiene `null`.

2.3 Constantes: el uso de *const* y *final*.

Si sabemos que el valor de una variable no va a cambiar en todo el programa es recomendable indicarlo en su declaración. Ello dará lugar a que el compilador pueda implementar una serie de optimizaciones en el tratamiento de estas variables que pueden dar lugar a código más eficiente. Este tipo de variables dejan, por tanto, de ser variables como tales. Nos referiremos a ellas como ***constantes***.

Podemos declarar constantes añadiendo cualquiera de los siguientes modificadores a la declaración de una variable:

a. `final`. Ofrece la posibilidad de declarar **constantes que se pueden inicializar en tiempo de ejecución**, por lo que no es necesario conocer su valor de antemano.

   No es obligatorio asignar el valor en la propia declaración de la constante, sino que puede hacerse a posteriori. Una vez le ha sido asignado un valor, ya no lo podremos cambiar.

   ```
   final DateTime hoy = DateTime.now();
   final int maxPaginas;           // No es obligatorio asignar el valor en la declaración
   maxPaginas = contarPaginas();   // Una vez establecido su valor, ya no lo podremos cambiar
   maxPaginas++;                   // Genera un error
   ```

b. `const`. Permite declarar **constantes puras, cuyo valor debe ser conocido en tiempo de compilación**, y debe ser proporcionado obligatoriamente en la declaración de la constante.

   ```
   const DateTime hoy = DateTime.now();  // Genera un error, ya que el valor solo podrá ser
                                         // resuelto en tiempo de ejecución
   const double numeroPI = 3.14;   // Asignamos obligatoriamente el valor en la declaración
   const int maxPaginas;    // Genera un error, ya que no hemos asignado un valor
   ```

Además de las diferencias comentadas, hay que tener en cuenta que el valor asignado a una `const` siempre será inmutable (no podrá cambiar). Con `final` esto si podría ocurrir en función del tipo de dato que se asigne. Veamos un ejemplo:

```
final valor = [1, 2, 3, 4];
print(valor); // Output: [1, 2, 3, 4]
valor.add(5); // Hacemos uso del método para agregar valor al array
print(valor); // Output: [1, 2, 3, 4, 5] (no genera error)
```

En cambio, con `const`:

```
const valor = [1,2,3,4];
print(valor); // Output: [1, 2, 3, 4]
valor.add(5); // Error Unhandled exception:  Unsupported operation
```

Por supuesto, podemos forzar a que la lista asignada como `final` no sea mutable haciendo lo siguiente:

```
final valor = const [1,2,3,4];
```

El uso correcto de `final` y `const` permite a Dart optimizar la ejecución de nuestro código a bajo nivel, y ahorrar recursos durante su ejecución de forma significativa.

2.4 Tipos de datos básicos y sus operadores.

Dart es un lenguaje orientado a objetos puro, eso significa que todos los datos que se usan en el lenguaje son objetos, y su correspondiente tipo de dato se considera una clase. Todos los tipos de datos heredan, en principio, de la clase `Object`, aunque matizaremos esto cuando hablemos de los tipos de datos *nullables*.

2.4.1 Booleanos.

Para representar valores booleanos (`true` o `false`) usaremos el tipo de dato `bool`.

2.4.2 Números.

En Dart tenemos dos tipos de datos disponibles para representar números: `int` (números enteros) y `double` (números con decimales). Ambos tipos heredan del *supertipo abstracto* `num`, que incluye varias operaciones y funciones matemáticas básicas comunes a números enteros y con decimales. De hecho, tanto `int` como `double` también son abstractos, ya que el compilador proporcionará la implementación concreta para estos tipos en cada plataforma.

2.4.3 Strings.

El tipo de dato `String` permite almacenar textos. Por ejemplo, el texto *Programadora full stack* se representaría como una lista de 23 caracteres:

P	r	o	g	r	a	m	a	d	o	r	a		f	u	l	l		s	t	a	c	k

Observa que el espacio en blanco también es un carácter y debe quedar representado dentro de la cadena.

El uso y manipulación de datos de tipo `String` es muy común. Por ello, prácticamente todos los lenguajes de programación facilitan esta labor en mayor o menor medida. En este sentido, con Dart estamos de suerte, ya que el trabajo con *strings* en este lenguaje es muy cómodo.

En Dart, los datos de tipo `String` se consideran valores atómicos o literales (al igual que los números y booleanos), y se escriben entre comillas simples `'` o dobles `"`. Veamos algunos ejemplos básicos:

```
// Declaramos algunas variables de tipo string?
String? miNombre, cadenaVacia, soyUnaCadena;

miNombre = 'Jose'; // Guardamos 'Jose' en la variable miNombre

cadenaVacia = '';  // Guardamos '' (cadena vacía) en la variable cadenaVacia
```

```
// Guardamos la cadena "15432" en la variable parezcoUnNumeroPeroSoyUnaCadena.
// Observa que este valor no es un número, sino un String.
// Por ejemplo, no podríamos dividirlo entre 2
String parezcoUnNumeroPeroSoyUnaCadena = '15432';
```

a) Concatenación.

Concatenar dos strings consiste simplemente en pegarlos tal cual. La forma más básica de hacer esto en Dart es mediante el uso del operador +, que al ser utilizado con operandos de tipo `String`, simplemente devolverá el resultado de su concatenación. Ejemplos:

```
String a = 'súper';
String b = 'guay';
String c = a + b;  // Se almacena 'súperguay' en la variable c
```

Si alguno de los operandos de + no es una cadena, se generará un error de compilación.

```
a = "El numero es" + 5;         // Genera error
b = 5 + " es el numero";        // Genera error
```

La forma de evitar el error sería la siguiente:

```
a = "El número es" + 5.toString();     // Genera la cadena 'El número es5'.
                                       // Observa que no hay espacio en blanco delante del 5
b = 5.toString() + " es el número";    // Genera la cadena '5 es el número', con un espacio en
                                       // blanco tras el 5, ya que se ha incluido explícitamente
                                       // en la cadena
```

A diferencia de JavaScript, Dart obliga a convertir en cadena todo aquello que queramos concatenar con el operador +, lo cual puede resultar bastante tedioso. Por suerte, existe una forma mucho más sencilla de hacer esto, tal y como vamos a ver a continuación

b) Interpolación de cadenas.

Cuando tenemos que construir cadenas a partir de datos que se encuentran en diferentes variables o que son resultado de determinadas expresiones, el uso de la concatenación puede tener un impacto negativo en la legibilidad de nuestro código y en su rendimiento.

Veamos un ejemplo en el que se resuelve una ecuación de segundo grado, y se muestra un mensaje final que se construye a partir de información que está dispersa en múltiples variables:

```
import 'dart:math';

void main(List<String> arguments) {
  double a = 1, b = 1, c = -2;
  double r = sqrt(b * b - 4 * a * c);
  double x1 = (-b + r) / (2 * a);
  double x2 = (-b - r) / (2 * a);

  String mensaje = 'La ecuación de segundo grado con coeficientes a=' +
      a.toString() + ', b=' + b.toString() + ' y c=' + c.toString() +
      ' tiene las soluciones x1=' + x1.toString() + ' y x2=' + x2.toString();

  print(mensaje);
```

```
}
```

La ejecución del ejemplo anterior reproducirá en la consola el siguiente mensaje:

```
La ecuación de segundo grado con coeficientes a=1, b=1 y c=-2 tiene las soluciones x1=1 y x2=-2
```

Como podemos observar, hay que hacer un uso intensivo del operador + que no solo complica mas de lo deseable la construcción del `String`, sino que da lugar a un código poco claro y difícil de leer. Para solucionar esto viene a nuestra ayuda la interpolación de cadenas.

La interpolación de cadenas facilita mucho la construcción de strings mediante el uso de *placeholders* (marcadores de posición). Sin más lio, voy a pasar directamente a reescribir el código del ejemplo anterior empleando interpolación de cadenas (omito la declaración e inicialización de las variables):

```
String mensaje = 'La ecuación de segundo grado con coeficientes a=$a, b=$b y c=$c tiene las soluciones x1=$x1 y x2=$x2';
```

Como podemos ver, en lugar de concatenar es mucho más práctico emplear *placeholders* dentro de la propia cadena.

Los *placeholders* se denotan con un `$` seguido de `{expresión}`. Al construir el `String`, Dart sustituirá los `${expresion}` que encuentre directamente por el valor de expresión. La expresión puede ser una simple variable (como lo que se hace en el ejemplo), o ser tan compleja como queramos. Si la *expresión* es una variable, no es necesario incluir las llaves `{}`.

Para ilustrar esto, vamos a modificar el ejemplo anterior para que el cálculo de las soluciones se realice directamente dentro de los propios placeholders, lo cual es perfectamente válido:

```
String mensaje = 'La ecuación de segundo grado con coeficientes a=$a, b=$b y c=$c tiene las soluciones x1=${(-b + r)/(2*a)} y x2=${(-b - r)/(2*a)}';
```

c) Cadenas de múltiples líneas.

En muchas ocasiones necesitamos construir *strings* que ocupan más de una línea de texto. De forma clásica, para ello había que utilizar el carácter salto de línea, denotado como `\n`, donde la `\` es un carácter especial que se denomina carácter de escape, del cual hablaremos a continuación.

Cuando construimos *strings*, el carácter `\` tiene un significado especial y hay que tratarlo con precaución. La `\` se denomina carácter de escape, y se usa para poder representar caracteres no imprimibles dentro de un `String`, como por ejemplo el salto de línea (`\n`) y el tabulador (`\t`). Así, si queremos que nuestro `String` contenga un salto de línea en un lugar determinado, escribiremos un `\n` en ese lugar. Si queremos una tabulación, escribiremos un `\t`.

¿Y qué pasa si queremos escribir una \ en la cadena? Pues para evitar que esta \ sea interpretada como carácter de escape debemos escribirla dos veces seguidas. De este modo, \\ se traduce a \ en nuestro `String` final. Otra posibilidad es poner una `r` (viene de *raw – crudo*) justo delante de la comilla de apertura del `String`:

```
String cadena = r'Soy una cadena con \n que no se interpreta como salto de línea';
```

Veamos un ejemplo de cadena de múltiples líneas basado de nuevo en el de la ecuación de segundo grado. Si queremos que el resultado de nuestro programa se muestre en un `String` de múltiples líneas como, por ejemplo, el siguiente:

```
\Ecuación de segundo grado\
Coeficientes:
     a=1, b=1, c=-2
Soluciones:
     x1=1, x2=-2
```

Lo podríamos construir así:

```
String mensaje = '\\Ecuación de segundo grado\\\nCoeficientes:\n\ta=$a, b=$b, c=$c\nSoluciones:\
n\tx1=$x1, x2=$x2';
```

Observa que `\n` se sustituye por un salto de línea, `\t` por una tabulación y `\\` por el carácter `\`. Por supuesto existe una manera mucho más cómoda y clara de hacer esto en Dart: **la triple comilla**, que permite construir cadenas de múltiples líneas y con caracteres especiales de forma mucho más cómoda e intuitiva. Haciendo uso de la triple comilla, nuestro `String` se podría escribir perfectamente así:

```
String mensaje = '''
\\Ecuación de segundo grado\\
Coeficientes:
     a=$a, b=$b, c=$c
Soluciones:
     x1=$x1, x2=$x2''';
```

2.4.4 Operadores.

a) Aritméticos.

Operador	Significado	Ejemplo
+	Suma dos valores	`5 + 4 // 9`
-	Resta dos valores	`3 - 5 // -2`
*	Multiplica dos valores	`4 * 5 // 20`
/	Divide dos valores	`11 / 2 // 5.5`
~/	División entera de dos valores	`11 ~/ 2 // 5`
%	Calcula el resto de una división entera	`11 % 2 // 1`

Los *prefijos y sufijos de incremento y decremento* también están disponibles, así como las *asignaciones aditivas*, *substractivas*, *multiplicativas* y *divisorias*. Veamos algunos ejemplos:

```
int miVar = 3;     // Declaramos la variable miVar, de tipo entero, inicializada a 3
miVar++;           // Devuelve el valor de a y luego lo incrementa
--miVar;           // Decrementa el valor de a y luego lo devuelve
miVar += 5;        // Asigna a miVar el resultado de sumar 5 a su valor, y lo devuelve
miVar -= 3;        // Asigna a miVar el resultado de restar 3 a su valor, y lo devuelve
miVar *= 2;        // Asigna a miVar el resultado de multiplicar por 2 su valor, y lo devuelve
miVar /= 2;        // Asigna a miVar el resultado de dividir entre 2 su valor, y lo devuelve
```

```
miVar %= 2;       // Asigna a miVar el resto de la división de su valor entre 2, y lo devuelve
```

b) Relacionales.

Operador	Significado	Ejemplo
==	Es igual que	`5 == 4 // false`
!=	Es distinto de	`3 != 5 // true`
>	Es mayor que	`4 > 5 // false`
<	Es menor que	`2 < 11 // true`
>=	Es mayor o igual que	`3 >= 3 // true`
<=	Es menor o igual que	`-3 <= 0 // true`

Además de números, estos operadores pueden aplicarse a otros tipos de datos. Por ejemplo, podemos aplicar el operador == para comprobar si dos cadenas de caracteres son iguales.

c) Lógicos.

Operador	Significado	Ejemplo
!	Cambia true a false, y viceversa	`!true // false`
&&	Devuelve true si ambos operandos son true	`true && false // false`
||	Devuelve true si al menos un operando es true	`-true || false // true`

Una observación importante con respecto al operador ! en Dart:

- Cuando lo usamos justo delante de una variable o expresión de tipo `bool`, devuelve su negación. Esto es similar a lo que sucede en otros lenguajes de programación. Ejemplo:

```
const esPar = 20 % 2 == 0;
const esImpar = !esPar;
```

- Sin embargo, y tal como vimos en el apartado 2.1.2.a, también puede ser usado **tras** una variable nullable, lo que producirá un intento de casting del valor de esta variable a un tipo no nullable, en tiempo de ejecución. Ejemplo:

```
int? value = 1;
int newValue = value!;
```

d) Condicionales.

Estos operadores permiten seleccionar entre uno de dos valores en base a si se cumple o no una condición determinada. Tenemos dos operadores con estas características en Dart:

- `valorA ?? valorB`

  Se devuelve `valorA` si no es `null`. En caso de que sea `null`, se devuelve `valorB`.

  Se conoce como el ***operador null-aware***, y se utiliza para proporcionar un valor de respaldo en caso de que una expresión sea nula. En el siguiente ejemplo, si valor no contuviese una cadena de caracteres que

representara a un entero válido, `tryParse()` devolvería `null`, y se asignaría un `0` a la variable `numero`. Ejemplo:

```
String valor = '42';
int numero = int.tryParse(valor) ?? 0;
print(numero);
```

- `condicion ? valorA : valorB`

  Si `condicion` es `true`, se devolverá `valorA`. En otro caso, se devolverá `valorB`.

  Se conoce como el operador ternario, y permite simplificar sentencias if sencillas. Ejemplo:

```
int edad = 20;
String mensaje = (edad >= 18) ? 'Eres mayor de edad' : 'Eres menor de edad';
print(mensaje);
```

3 Funciones.

Las funciones en Dart permiten encapsular lógica y reutilizar código. Además, el uso de parámetros opcionales, ya sean con nombre o posicionales, brinda flexibilidad al permitir que las funciones se llamen con diferentes conjuntos de argumentos.

3.1 Declaración y uso de funciones.

Aquí tienes un ejemplo de declaración y uso básico de una función en Dart:

```
void saludar() {
  print('¡Hola!');
}
```

En este ejemplo, declaramos una función `saludar()` que no recibe ningún parámetro y no devuelve ningún valor (`void`). Dentro del bloque de la función, simplemente imprimimos `'¡Hola!'`. Luego, en la función `main()`, llamaremos a la función `saludar()` utilizando el nombre de la función seguido de paréntesis `()`:

```
void main() {
  saludar(); // Llamada a la función saludar
}
```

La salida en la consola será:

```
¡Hola!
```

3.2 Funciones con parámetros.

Las funciones admiten parámetros en su declaración. Por ejemplo, podemos modificar la función `saludar()` para que reciba un parámetro `nombre` que contenga el nombre de la persona a saludar:

```
void saludar(String nombre) {
  print('¡Hola, $nombre!');
}
```

En este ejemplo, la función `saludar` tiene un parámetro llamado `nombre` de tipo `String`. Al llamar a la función, pasamos un valor específico para ese parámetro, tal y como se muestra a continuación:

```
void main() {
  saludar('Paco');
  saludar('Marta');
}
```

La salida en la consola será:

```
¡Hola, Paco!
¡Hola, Marta!
```

En cada llamada a la función `saludar()`, el parámetro `nombre` toma el valor que le pasamos como argumento y se imprime un saludo utilizando ese valor.

Podemos definir funciones que reciban tantos parámetros como queramos, sin más que indicarlos en la declaración de la función separados por comas.

3.3 Funciones que devuelven un valor.

Una función puede devolver un valor utilizando la palabra clave `return` seguida del valor que se desea retornar. La declaración del tipo del valor retorno se puede especificar antes del nombre de la función. En este ejemplo, puedes ver una función `suma()` que acepta dos parámetros de tipo `int`. Dentro del cuerpo de la función, se utiliza la expresión `return` para devolver la suma de los dos parámetros `a` y `b`. En este punto, la ejecución de la función finaliza y el valor especificado se devuelve al punto de llamada.

```
int suma(int a, int b) {
  return a + b;
}
```

Puedes asignar el valor devuelto de una función a una variable, pasarlo como argumento a otra función o realizar cualquier otra operación que desees. Ejemplo:

```
void main() {
  int resultado = suma(3, 5);
  print(resultado);
}
```

La salida en la consola será:

```
8
```

3.4 Funciones con parámetros opcionales.

Los parámetros que se definen en la lista de parámetros de una función sin ninguna notación especial, es decir, sin corchetes o llaves, son parámetros obligatorios y deben ser proporcionados al llamar a la función en el orden correcto.

Sin embargo, además de los parámetros obligatorios, Dart permite el uso de parámetros opcionales. Hay dos tipos de parámetros opcionales: los parámetros con nombre y los parámetros posicionales opcionales.

- **Parámetros con nombre.** Puedes utilizar los parámetros con nombre para especificar los argumentos que deseas pasar a una función utilizando su nombre. Se declaran dentro de llaves `{}` en la lista de parámetros de la función, y se les asigna un valor por defecto utilizando el operador :. Aquí tienes un ejemplo:

  ```
  void saludar({String nombre = 'Invitado'}) {
    print('¡Hola, $nombre!');
  }
  ```

  El valor del parámetro opcional con nombre se proporcionaría del siguiente modo

  ```
  void main() {
    saludar(nombre: 'Alicia');
  }
  ```

La salida en la consola será:

```
¡Hola, Alicia!
```

Si queremos que alguno de los parámetros con nombre sea obligatorio, podemos incluir delante de él la palabra `required`.

- **Parámetros posicionales opcionales.** Se declaran utilizando corchetes `[]` en la lista de parámetros de la función. Estos parámetros pueden ser omitidos al llamar a la función y se permite definir un valor por defecto para los mismos. Aquí tienes un ejemplo:

```dart
void saludar(String nombre, [int edad = 0]) {
  print('¡Hola, $nombre! Tienes $edad años.');
}
```

Los siguientes dos usos de `saludar()` son perfectamente válidos:

```dart
void main() {
  saludar('Bob');
  saludar('Charlie', 30);
}
```

3.5 Funciones arrow.

En Dart, las ***funciones arrow*** (o funciones flecha, también conocidas como funciones lambda) son una sintaxis alternativa que permite definir funciones de forma más concisa. Veamos un ejemplo:

```dart
int calcularCuadrado(int numero) => numero * numero;
```

En este ejemplo, la función flecha `calcularCuadrado()` toma un argumento `numero` y devuelve el resultado de multiplicar `numero` por sí mismo. No es necesario utilizar la palabra clave `return` explícitamente, ya que el resultado se devuelve implícitamente. Tampoco es necesario declarar el tipo devuelto por la función, ya que puede ser inferido por el compilador a partir del tipo de los valores devueltos.

Por ejemplo, lo siguiente también sería válido:

```dart
calcularCuadrado(int numero) => numero * numero;
```

Una gran ventaja de las funciones arrow es que facilitan mucho el trabajo con ***funciones de orden superior***, que son aquellas que reciben alguna función por parámetro. Ejemplos de funciones (métodos en este caso) de orden superior los tenemos en los métodos `map()`, `forEach()` y `where()` de las listas. Estas, en concreto, aplican la función que reciben a cada elemento de una lista.

En el siguiente ejemplo, la función flecha `(numero) => numero * numero` se pasa como argumento al método `map()` y se aplica a cada elemento de la lista `numeros`, para construir una nueva lista con los todos los resultados devueltos. El resultado es una nueva lista `cuadrados` que contiene los cuadrados de los números originales:

```dart
List<int> numeros = [1, 2, 3, 4, 5];
List<int> cuadrados = numeros.map((numero) => numero * numero).toList();
// cuadrados valdrá [1, 4, 9, 16, 25]
```

Observa como en este caso, ni siquiera es necesario asignar un nombre a la función `arrow`. Es lo que se llama una función anónima.

3.6 El tipo de dato Function.

El tipo de dato Function es muy útil porque nos permite declarar variables y parámetros que aceptan como tipo de dato una función. Por ejemplo, podemos declarar una variable `miFun` cuyo tipo de datos es una función que toma dos números enteros y devuelve un número entero. La sintaxis sería la siguiente:

```
int Function(int, int) miFun;         // Variable miFun, de tipo int Function(int, int)
```

Observa que estamos declarando una variable `miFun` , y que su tipo es `int Function(int, int)`. Seguidamente le podemos asignar la función `calcularCuadrado`, cuya signatura encaja con el tipo de función que acepta la variable `miFun`:

```
miFun = calcularCuadrado;
```

Por supuesto, podemos asignarle una función arrow anónima:

```
miFun = int (int a, int b) => a + b;
```

Si queremos invocar la función asignada a `miFun`:, no hay que hacer nada en especial, simplemente tenemos que llamar a `miFun` tal y como hacemos con cualquier otra función:

```
int resultado = miFun(2, 5);
print(resultado);
```

El tipo de dato `Function` también **nos permite implementar nuestras propias funciones de orden superior**, como es el caso del siguiente ejemplo, donde definimos una función aplicar que recibe una función y dos valores por parámetro, y devuelve el resultado de invocar la función recibida sobre esos dos valores. Se han marcado los parámetros individualmente con fondo amarillo, para que se distingan con claridad:

```
double aplicar(double Function(double, double) f, double a, double b){
  return f(a, b);
}

double resultado2 = aplicar( (p1, p2)=>(p1 + p2)/2 , 10.6, 15.2 );
print(resultado2);
```

4 Sentencias de la programación estructurada.

4.1 Bifurcación.

4.1.1 La sentencia if.

En Dart, la sentencia `if` se utiliza para controlar el flujo de ejecución de un programa en función de una condición. Veamos un ejemplo básico con una sola condición:

```
if (edad >= 18) {
  print("Eres mayor de edad. Puedes ingresar al sitio.");
}
```

En este caso, la condición es `edad >= 18`. Si la edad es mayor o igual a `18`, se ejecutará el código dentro del bloque `{}`, que en este ejemplo imprime *Eres mayor de edad. Puedes acceder*.

Ahora, supongamos que queremos agregar una condición adicional utilizando `else if`. Por ejemplo:

```
if (edad >= 18) {
  print("Eres mayor de edad. Puedes acceder.");
} else if (edad >= 13) {
  print("Eres adolescente. Puedes acceder con limitaciones.");
}
```

Aquí, si la primera condición `edad >= 18` es falsa, se evalúa la segunda condición `edad >= 13`. Si la edad es mayor o igual a `13` pero menor que `18`, se ejecutará el código dentro del bloque {}, que en este caso imprimirá *Eres adolescente. Puedes acceder a ciertos contenidos*.

Finalmente, también podemos agregar un bloque `else` para manejar cualquier otro caso que no cumpla con las condiciones anteriores. Por ejemplo:

```
if (edad >= 18) {
  print("Eres mayor de edad. Puedes acceder.");
} else if (edad >= 13) {
  print("Eres adolescente. Puedes acceder con limitaciones.");
} else {
  print("Eres menor de edad. No puedes acceder.");
}
```

4.1.2 La sentencia switch.

La sentencia switch en Dart se utiliza para tomar decisiones basadas en múltiples casos posibles de una variable o expresión. Proporciona una forma más estructurada de manejar múltiples opciones en comparación con una serie de sentencias `if-else`. Ejemplo:

```
String diaDeLaSemana = 'Lunes';
switch (diaDeLaSemana) {
  case 'Lunes':
    print('Hoy es lunes. Toca comenzar la semana.');
  case 'Martes':
    print('Hoy es martes. Sigamos adelante.');
  case 'Miércoles':
    print('Hoy es miércoles. Mitad de semana.');
```

```
  case 'Jueves':
    print('Hoy es jueves. Se acerca el fin de semana.');
  case 'Viernes':
    print('Hoy es viernes. ¡Fin de semana a la vista!');
  default:
    print('Es otro día de la semana.');
}
```

En este ejemplo, tenemos una variable `diaDeLaSemana` que contiene el día actual. La sentencia `switch` evalúa el valor de `diaDeLaSemana` y ejecuta el bloque de código correspondiente al caso que coincida. Veamos cómo funciona:

- Si `diaDeLaSemana` es `'Lunes'`, se ejecutará el código dentro de `case 'Lunes'`, que imprimirá *Hoy es lunes. Toca comenzar la semana.*
- Si `diaDeLaSemana` es `'Martes'`, se ejecutará el código dentro de `case 'Martes'`, que imprimirá *Hoy es martes. Sigamos adelante.*
- Y así sucesivamente. Si `diaDeLaSemana` no coincide con ninguno de los casos, se ejecutará el bloque `default`, que imprimirá *Es otro día de la semana.*

A diferencia de otros lenguajes de programación, en Dart no es obligatorio el uso de `break` dentro de cada caso de `switch`, aunque está disponible para cuando lo necesitemos. Para cada case se ejecutará solo el código que haya hasta el siguiente `case`, aunque `break` haya sido omitido. Una excepción a esto es cuando aparecen varios `case` juntos:

```
String diaDeLaSemana = 'Lunes';
switch (diaDeLaSemana) {
  case 'Lunes':
  case 'Martes':
  case 'Miércoles':
    print('Estamos en la primera mitad de la semana');
  case 'Jueves':
    print('Hoy es jueves. Se acerca el fin de semana.');
  case 'Viernes':
    print('Hoy es viernes. ¡Fin de semana a la vista!');
  default:
    print('Es otro día de la semana.');
}
```

4.2 Iteración.

4.2.1 El bucle for.

El bucle `for` permite repetir un bloque de código un número específico de veces o recorrer elementos de una lista, utilizando una variable de control. Proporciona una forma conveniente de iterar sobre una secuencia de valores y ejecutar ciertas instrucciones en cada iteración. Ejemplo:

```
for (var i = 1; i <= 5; i++) {
  print('El valor de i es $i');
}
```

En este ejemplo, el bucle for se utiliza para imprimir los valores del `1` al `5`. Veamos cómo funciona:

- `var i = 1`
  Se declara una variable `i`, inicializándola a `1`. Esta variable se utiliza como contador para controlar el número de iteraciones.
- `i <= 5`
  Esta es la condición que se evalúa antes de cada iteración. Mientras la condición sea verdadera, el bucle continuará ejecutando su bloque de código.
- `i++`
  Después de cada iteración, se incrementa el valor de `i` en `1`.
- `print('El valor de i es $i')`
  Este es el bloque de código que se ejecuta en cada iteración. Imprime el valor actual de `i`.

En cada iteración del bucle, el valor de `i` se incrementa y se imprime en la consola. La salida será la siguiente:

```
El valor de i es 1
El valor de i es 2
El valor de i es 3
El valor de i es 4
El valor de i es 5
```

4.2.2 El bucle while.

El bucle while se utiliza para repetir un bloque de código mientras se cumpla una condición específica. Este bucle ejecuta el bloque de código repetidamente siempre que la condición sea verdadera y finaliza cuando la condición se evalúa como falsa. Ejemplo:

```
int contador = 1;
while (contador <= 5) {
  print('El valor del contador es $contador');
  contador++;
}
```

En este ejemplo, se tiene una variable `contador` que se inicializa con el valor `1`. El bucle `while` se repite mientras la condición `contador <= 5` sea verdadera. En cada iteración del bucle, el bloque de código dentro del bucle se ejecuta, que en este caso imprime el valor actual del `contador`. Luego, se incrementa el valor de `contador` en `1` usando el operador `++`. Tras ejecutar este bucle aparecerá el siguiente resultado por la consola:

```
El valor del contador es 1
El valor del contador es 2
El valor del contador es 3
El valor del contador es 4
El valor del contador es 5
```

Es importante tener en cuenta que debemos asegurarnos de que la condición del bucle `while` en algún momento llegue a evaluarse como `falsa`, para así evitar un bucle infinito. En este caso, la condición `contador <= 5` se dejará de cumplirse cuando `contador` alcance el valor `6`, y el bucle se detendrá.

El bucle `while` es útil cuando no sabemos exactamente cuántas iteraciones se necesitarán, pero se requiere que el bloque de código se ejecute mientras se cumpla una condición específica.

4.2.3 El bucle do-while.

Es similar al bucle `while`, pero con una diferencia fundamental: la condición se evalúa al final del ciclo, después de ejecutar el bloque de código. Esto significa que el bloque de código se ejecutará al menos una vez, independientemente de si la condición es verdadera o falsa.

Aquí tienes el ejemplo anterior escrito con un bucle do-while:

```
int contador = 1;
do {
  print('El valor del contador es $contador');
  contador++;
} while (contador <= 5);
```

5 Clases.

Una clase es una plantilla que define el comportamiento y las propiedades de un objeto. En Dart, puedes definir clases utilizando la palabra clave `class`. A lo largo de este apartado nos basaremos en el siguiente ejemplo para explicar las características de las clases en Dart:

```dart
class Persona {
  String nombre;          // Campo público, de tipo string no nulable
  int edad;               // Campo público, de tipo int no nulable
  String? direccion;      // Campo público, de tipo string nulable
  String _dni;            // Campo privado (_ delante del nombre), de tipo string no nulable

  // Constructor
  Persona(this.nombre, this.edad, String dni, String? dir) // Lista de parámetros con sintaxis
    : this._dni = dni     // Bloque de inicialización       // de inicialización formal (this.)
  {                       // Cuerpo del constructor
    direccion = dir;
  }

  // Métodos
  void saludar() {
    print('Hola, mi nombre es $nombre');
  }

  void cumplirAnios() {
    edad++;
    print('¡Feliz cumpleaños! Ahora tengo $edad años');
  }

  @override
  String toString() => '$nombre, $edad';
}
```

Podemos usar la clase `Persona` para crear objetos usando para ello su constructor, tal y como se muestra en el siguiente ejemplo:

```dart
void main() {
  Persona p = Persona('Juan', 25); // No es necesario new para crear nuevos objetos
  p.saludar(); // Salida: Hola, mi nombre es Juan
  p.cumplirAnios(); // Salida: ¡Feliz cumpleaños! Ahora tengo 26 años
}
```

Como puedes observar, hemos creado una instancia (objeto) de `Persona` con el nombre `"Juan"` y edad `25`. Luego, hemos invocado sus métodos `saludar()` y `cumplirAnios()`, obteniendo los mensajes correspondientes en la consola.

5.1 Miembros de una clase.

Los miembros de una clase pueden ser campos (también denominados propiedades) o métodos (similares a las funciones). Si observamos la clase `Persona`, podemos ver que tiene cuatro propiedades (se declaran de forma similar a las variables),

```
  String nombre;          // Campo público, de tipo string no nulable
  int edad;               // Campo público, de tipo int no nulable
  String? direccion;      // Campo público, de tipo string nulable
  String _dni;            // Campo privado (_ delante del nombre), de tipo string no nulable
```

un constructor (función con el mismo nombre de la clase),

```
  // Constructor
  Persona(this.nombre, this.edad, String dni, String? dir) // Lista de parámetros con sintaxis
    : this._dni = dni     // Bloque de inicialización       // de inicialización formal (this.)
  {                       // Cuerpo del constructor
    direccion = dir;
  }
```

y tres métodos:

- `saludar()`. Imprime un mensaje de saludo en la consola utilizando el nombre de la persona.
- `cumplirAnios()`. Incrementa la edad en uno y muestra un mensaje de feliz cumpleaños.
- `toString()`. Devuelve un `String` con los datos de la persona. Observa que este método tiene la anotación `@override`, que denota que el método es una implementación que sustituye al método `toString()` de la clase padre (`Object`, en este caso).

Con respecto a los métodos, decir que su declaración se realiza de forma idéntica a las funciones vistas en el apartado 3.

5.2 El uso de this.

En Dart no necesitamos usar el prefijo `this` para acceder a los miembros de una clase desde los métodos y el constructor de la propia clase. Echemos un vistazo al método `cumplirAnios()` para comprobarlo:

```
void cumplirAnios() {
  edad++;
  print('¡Feliz cumpleaños! Ahora tengo $edad años');
}
```

Como puedes ver, se accede al campo edad sin necesidad de `this`. Sin embargo, hay situaciones donde esto es necesario, sobre todo por ambigüedad entre algún parámetro o variable local, y el campo en la clase. Ejemplo:

```
void cambiarNombre(String nombre){
  this.nombre = nombre; // Usamos this para diferenciar el campo nombre del parámetro nombre
}
```

5.3 El constructor.

El constructor es una función particular que se invoca para crear objetos de la clase. En Dart, el nombre del constructor debe coincidir con el nombre de la clase, pueden recibir parámetros, pueden tener un cuerpo (no es obligatorio), e incluso una sección de inicialización. Con respecto a la lista de parámetros, decir que se aplican las mismas reglas vistas para la declaración de funciones, pero con algunas novedades que examinaremos a lo largo de este apartado. Por cierto, no es obligatorio implementar un constructor para una clase, en cuyo caso el compilador de Dart proporcionará un constructor sin parámetros y sin cuerpo.

Sin embargo, es importante tener en cuenta que en Dart, la inicialización de campos no nulables no puede realizarse en el cuerpo del constructor, siendo necesario recurrir sintaxis especial para ello. Por ejemplo, el siguiente código, aparentemente normal en cualquier lenguaje de programación, generará un error de compilación en Dart:

```
Persona(int nombre, String edad, String dni, String? dir)
{
  this.nombre = nombre; // Los campos no nulables no pueden inicializarse así en Dart
  this.edad = edad;
  this._dni = dni;
  direccion = dir; // dirección es nulable, así que esta forma de inicializar su valor no
                   // genera ningún problema
}
```

La forma correcta de inicializar estas propiedades la podemos ver en el siguiente fragmento de código, correspondiente al constructor de la clase `Persona`:

```
// Constructor
Persona(this.nombre, this.edad, String dni, String? dir):this._dni = dni {
  direccion = dir;
}
```

En este fragmento de código podemos ver que en el constructor se utiliza la *sintaxis de inicialización formal*, en combinación con un *bloque de inicialización*, para establecer el valor inicial de las propiedades nulables del objeto. Analizaremos estas dos maneras de inicializar propiedades a continuación.

5.3.1 Sintaxis de inicialización formal.

Esta sintaxis es un dos en uno: permite declarar parámetros e inicializar propiedades a la vez, de forma concisa, mediante el uso de `this` en la lista de parámetros del constructor. Por ejemplo, al incluir `this.nombre` en la lista de parámetros, estamos indicando lo siguiente:

- El constructor recibe el parámetro `nombre`, del mismo tipo que el de la propiedad con igual nombre, declarada en la clase.
- El valor que se reciba para este parámetro se usará para inicializar esa propiedad.

En la clase `Persona`, hemos usado este mecanismo en el constructor para inicializar las propiedades `nombre` y `edad`.

5.3.2 Bloque de inicialización.

Permite inicializar los campos privados de forma explícita mediante asignaciones, justo antes del cuerpo del constructor. El bloque de inicialización se declara añadiendo dos puntos tras el constructor, y seguidamente realizando las asignaciones correspondientes, separadas por comas. Veámos un ejemplo:

```
Persona(int nombre, String edad, String dni, String? dir):
  this.nombre = nombre, this.edad = edad, this._dni = dni
{
  direccion = dir;
}
```

Esto es particularmente útil cuando queremos inicializar campos privados a través del constructor, pero no queremos usar un nombre de parámetro con una barra baja. Este es el caso del campo `_dni`, que en la definición original de la clase es inicilizado mediante este mecanismo.

Ambos métodos son compatibles, pudiéndo combinarlos según convenga, tal y como hemos hecho en el constructor original de la clase Persona:

```
// Constructor
Persona(this.nombre, this.edad, String dni, String? dir) : this._dni = dni {
  direccion = dir;
}
```

5.4 Miembros públicos, privados y protegidos.

A nivel de lenguaje, Dart solo soporta miembros ***públicos*** y ***privados***. A este respecto, todos los miembros de una clase son públicos, salvo que su nombre comience por una barra baja _, en cuyo caso se trataran como miembros privados.

Sin embargo, la librería *meta*[2] ofrece una colección de anotaciones que los desarrolladores pueden usar para expresar intenciones o características que no se pueden deducir del análisis estático del código. Entre ellas podemos encontrar la anotación `@protected`, que añadida justo delante del miembro de una clase hace que el analizador lo trate como ***protegido***, de modo que ese miembro solo será accesible dentro de la jerarquía de herencia. Por ejemplo, si queremos que la visibilidad del campo `nombre` sea *protegida*, bastaría con colocar la anotación `@protected` justo delante de él:

```
@protected String nombre;
```

Es importante tener en cuenta que para hacer uso de estas anotaciones es necesario incluir la librería meta como dependencia del proyecto. Para ello editamos el archivo `pubspec.yaml`, y modificamos la sección de dependencias como se indica:

```
# Add regular dependencies here.
dependencies:
  meta: ^1.9.1 # Observa que hemos declarado una dependencia hacia la version 1.9.1 de meta
```

Al guardar los cambios, la extensión de Dart de VSCode descargará e incluirá la librería en nuestro proyecto. Ahora solo queda importar la librería *meta* dentro del archivo `.dart` donde queremos usar la anotación. Para ello añadimos la siguiente línea al principio del archivo:

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

5.5 Miembros estáticos.

2 https://pub.dev/documentation/meta/latest/meta/meta-library.html

Denominamos miembros estáticos a las propiedades y métodos declarados en una clase, que pertenecen a la clase en sí misma, en lugar de a instancias individuales de la clase. Los miembros estáticos se pueden usar para almacenar datos o definir comportamientos que son comunes a todas las instancias de la clase. Para definir un miembro como estático, basta con marcarlo con la palabra clave `static`. Podemos tener tanto propiedades (campos) como métodos estáticos:

1. ***Campos estáticos***. Un campo estático se declara utilizando la palabra clave `static` antes de su tipo. Estos campos existen independientemente de cualquier instancia de la clase y se comparten entre todas las instancias. Pueden ser accedidos directamente utilizando el nombre de la clase seguido de un punto, y el nombre del campo.

```
class MiClase {
  static int miVariable = 10;
}

void main() {
  print(MiClase.miVariable);  // Imprime: 10
}
```

2. ***Métodos estáticos***. Un método estático se declara utilizando la palabra clave `static` antes del tipo de retorno o `void`. Estos métodos no tienen acceso a las variables de instancia de la clase, pero pueden acceder a otras variables estáticas. Al igual que las variables estáticas, los métodos estáticos se pueden llamar directamente utilizando el nombre de la clase seguido de un punto y el nombre del método.

```
class MiClase {
  static void miMetodo() {
    print('Este es un método estático');
  }
}

void main() {
  MiClase.miMetodo();  // Imprime: Este es un método estático
}
```

A continuación puedes ver un ejemplo de clase que implementa miembros estáticos:

```
class MathUtils {
  static const double pi = 3.14159;

  static double calcularAreaCirculo(double radio) => pi * radio * radio;

  static double calcularPerimetroCuadrado(double lado) => 4 * lado;
}

void main() {
  double area = MathUtils.calcularAreaCirculo(5.0);
  double perimetro = MathUtils.calcularPerimetroCuadrado(8.0);
}
```

Observa que desde un método estático solo podemos acceder a miembros estáticos de la clase. Por ejemplo, desde el método estático `calcularAreaCirculo` se puede acceder a la constante `pi`, ya que también es estática.

5.6 Constructores especiales.

5.6.1 Constructores con nombre.

Podemos añadir múltiples constructores a una misma clase mediante el uso de ***constructores con nombre***. Veámoslo con un ejemplo:

```
class Persona {
  String nombre;
  int edad;

  Persona(this.nombre, this.edad); // Constructor por defecto

  Persona.nacimiento(this.nombre) {
    // Constructor generativo para recién nacidos
    edad = 0;
  }

  Persona.adulto(this.nombre) {
    // Constructor generativo para adultos
    edad = 18;
  }

  String saludar() {
    return 'Hola, soy $nombre y tengo $edad años.';
  }
}
```

Como puedes observar, la clase Persona ahora tiene tres constructores generativos: al constructor por defecto se le unen los constructores `Persona.nacimiento()` y `Persona.adulto()`. Estos constructores se usarían del siguiente modo:

```
void main() {
  var juan = Persona('Juan', 25); // Constructor por defecto
  juan.saludar();

  var bebe = Persona.nacimiento('Luisa'); // Constructor generativo para recién nacidos
  bebe.saludar();

  var adulto = Persona.adulto('María'); // Constructor generativo para adultos
  print(adulto.saludar());
}
```

5.6.2 Constructores privados.

Podemos declarar ***constructores privados***. Para ello solo tenemos que aplicar el mecanismo visto en el apartado 6.4 para declarar miembros privados: basta con hacer que el nombre del constructor quede precedido por un carácter _. En caso de que queramos que el constructor por defecto sea privado, haremos que su nombre sea un carácter _ aislado. Ejemplo:

```
class ClaseNoInstanciable{
  ClaseNoInstanciable._
}
```

Al declarar un constructor como privado, hacemos que su uso solo sea posible dentro de la propia clase, desde otro constructor o método. Veremos algunos casos de uso frecuentes de constructores privados en el siguiente apartado.

5.6.3 Constructores generativos y constructores factory.

Los constructores vistos hasta ahora se denominan ***constructores generativos***, ya que generan una nueva instancia de la clase cada vez que son invocados. Sin embargo, existe otro tipo de constructores en Dart: los constructores factory.

Los ***constructores factory*** son un tipo especial de constructor que no tienen esta limitación; es decir, ofrecen control sobre qué instancia de la clase se debe devolver. Un caso de uso común para este tipo de constructores es la implementación del patrón *Singleton*, que también requiere el uso de un constructor privado.

El ***patrón Singleton*** es un patrón de diseño que se utiliza para garantizar que solo exista una única instancia de una clase en todo el programa. Su objetivo principal es restringir la creación de instancias adicionales y proporcionar un punto global de acceso a la instancia existente.

Veamos un ejemplo de implementación de este patrón mediante un constructor generativo privado, y haciendo que el constructor por defecto sea de tipo *factory*:

```dart
class UnaClaseSingleton {
  static UnaClaseSingleton _instance;

  factory UnaClaseSingleton() {
    if (_instance == null) {
      _instance = UnaClaseSingleton._interno();
    }
    return _instance;
  }

  // Hacemos que el constructor por defecto sea privado,
  // usando un constructor cuyo nombre comienza por _
  UnaClaseSingleton._interno() {
    // Lógica de inicialización del objeto
  }

  void saludar() {
    print('Hola, soy un singleton.');
  }
}

void main() {
  var singleton1 = UnaClaseSingleton();
  var singleton2 = UnaClaseSingleton();

  print(identical(singleton1, singleton2)); // true: se trata del mismo objeto

  singleton1.saludar();
  singleton2.saludar();
}
```

Otro caso de uso de un constructor factory lo tenemos cuando se necesita ***computar el valor inicial de algún campo no nulable***, y esto no puede hacerse directamente en el constructor generativo.

Ejemplo:

```
import 'dart:math';

class TrianguloRectangulo {
  double cateto1;
  double cateto2;
  double hipotenusa;

  TrianguloRectangulo._(this.cateto1, this.cateto2, this.hipotenusa);

  factory TrianguloRectangulo(double cateto1, double cateto2) {
    final hipotenusa = sqrt(pow(cateto1, 2) + pow(cateto2, 2));
    return TrianguloRectangulo._(cateto1, cateto2, hipotenusa);
  }

  void mostrarInfo() {
    print('Cateto1: ${cateto1.toStringAsFixed(2)}');
    print('Cateto2: ${cateto2.toStringAsFixed(2)}');
    print('Hipotenusa: ${hipotenusa.toStringAsFixed(2)}');
  }
}

void main(List<String> arguments) {
  TrianguloRectangulo t = TrianguloRectangulo(3, 4);
  t.mostrarInfo();
}
```

En este caso, aunque el constructor por defecto es factory, siempre que se invoca genera una nueva instancia de la clase. Su utilidad reside en que permite computar un valor, para seguidamente pasar este valor al constructor generativo, donde será usado para inicializar un miembro no nulable sin ningún problema.

5.6.4 Constructores const.

Los constructores const en Dart son constructores que crean objetos constantes, no mutables. Un objeto constante es un objeto que no puede ser modificado después de su creación. Los constructores `const` se utilizan a menudo para crear objetos que deben ser inmutables, como cadenas o números.

Una clase puede tener un constructor `const` solo si todos los campos de la clase son finales. Para crear un constructor `const`, debe usar la palabra clave `const` antes del nombre del constructor. Por ejemplo, el siguiente código crea un constructor const para una clase llamada `Point`:

```
class Point {
  const Point(this.x, this.y);
  final int x;
  final int y;
}
```

El constructor `Point` puede utilizarse para crear objetos constantes de la siguiente manera:

```
const pointA = Point(1, 2); // Observa que point es declarado como constante
```

También podemos asignar el objeto constante a una variable del siguiente modo:

```
Point pointB = const Point(1, 2);
```

Tanto `pointA` como `pointB` hacen referencia a un objeto constante, inmutable, cuyo estado no puede ser modificado después de su creación. Por ejemplo, el siguiente código producirá un error:

```
point.x = 3;
```

La diferencia es que a `pointB` podríamos asignarle otra referencia diferente en tiempo de ejecución (es una variable). Ejemplo:

```
pointB = const Point(3,4);
```

El uso de constructores const puede dar lugar a una mejora significativa en el rendimiento de nuestra aplicación, ya que:

- Permiten al compilador de Dart aplicar una serie de optimizaciones, creados estos objetos en tiempo de compilación.
- Producen un mayor aprovechamiento de la memoria, ya que si tenemos dos o más objetos const iguales en diferentes partes de nuestro programa, el compilador de Dart los detectará y usará una única instancia común en memoria para todos ellos.

No obstante, el uso de `const` en el constructor no es siempre posible, ya que es perfectamente normal que el estado de un objeto pueda cambiar. Sin embargo, si sabemos que los objetos no van a ser mutables, debemos implementar constructores const para sus clases correspondientes, ya que con ello obtendremos un mayor rendimiento en nuestra aplicación.

5.7 Herencia.

La herencia es un concepto fundamental en la programación orientada a objetos que permite que una clase adquiera las propiedades y comportamientos de otra clase, conocida como la clase padre o *superclase*. La clase que hereda estas características se denomina clase hija o subclase.

Dart soporta herencia simple; es decir, una clase solo puede heredar de una sola clase a la vez. La sintaxis básica para indicar que una clase hereda de otra es la siguiente:

```
class ClasePadre {
  // Propiedades y métodos de la clase padre
}

class ClaseHija extends ClasePadre {
  // Propiedades y métodos de la clase hija
}
```

A través de la herencia, la clase hija puede acceder a los miembros públicos y protegidos de la clase padre. Los miembros privados de la clase padre no son accesibles directamente por la clase hija. Por ejemplo, vamos a definir una clase `Vehiculo` con las siguientes características:

```
class Vehiculo {
  int nRuedas;
  int nPlazas;

  Vehiculo(this.nRuedas, this.nPlazas);

  void aparcar() {
    print("El vehículo ha sido aparcado.");
  }
}
```

A continuación, definiremos una clase `Coche` que heredará de `Vehículo`, adquiriendo todas sus propiedades y métodos, a los que añadirá algunos miembros más:

```
class Coche extends Vehiculo {
  String matricula;
  final bool _esCombustion;
  final bool _esElectrico;

  Coche(this._esCombustion, this._esElectrico, this.matricula) : super(4, 5);

  @override
  void aparcar() {
    print("El coche ha sido aparcado.");
  }

  String get tipoMotor => (_esCombustion && _esElectrico)
      ? "Hibrido"
      : (_esCombustion ? "Combustion" : "Eléctrico");
}
```

En este ejemplo, la clase `Coche` hereda de la clase `Vehiculo`. Al crear un objeto de la clase `Coche`, podemos acceder tanto a las propiedades y métodos definidos en la clase `Coche`, como a los heredados de la clase `Vehiculo`.

Es importante el uso que se hace de `super()` en el constructor de `Coche`, cuya finalidad es la de realizar una llamada al constructor de la clase padre `Vehiculo` que la inicialice correctamente.

5.8 Aspectos relativos a la nulabilidad.

Debemos tener precaución a la hora de tratar con variables nulables o parámetros nulables que almacenan referencias a objetos. Por ejemplo, partimos de una sencilla función como la siguiente, donde se recibe un parámetro de tipo `Persona?`:

```
void mostrarSaludo(Persona? persona){
  print(persona.saludar());     // Error de compilación: se llama a un método de un objeto
}                               // persona que podría ser null
```

En primer lugar, vemos que `p.toString()` generará error de compilación, ya que `p` es nulable. Para solucionar este error de compilación, podríamos vernos tentados a realizar la siguiente comprobación con la sentencia `if`:

```
if (persona){                   // Condiciones como esta pueden funcionar en otros lenguajes
  print(persona.saludar());    // pero no en Dart, que generará un error de compilación
}
```

Este tipo de condiciones no son válidas en Dart. Para hacerlo correctamente, debemos comparar explícitamente con `null`:

```
if (persona != null){
  // Aquí dentro podemos asumir que persona es no nulable y ya no tendremos errores de
  // compilación relativos a su nulabilidad
  print(persona.saludar());
}
```

Al realizar la comprobación para asegurar que persona es distinto de `null`, el compilador asumirá que dentro del bloque del `if` la variable `persona` es *no nulable*, y podremos acceder a sus métodos y propiedades sin problema.

Todo esto está muy bien, pero tener que escribir una sentencia `if` cada vez que queremos acceder a un método o campo de un objeto que podría ser `null` es un poco incómodo y tedioso. Por ello, Dart ofrece ciertas facilidades a la hora de trabajar con referencias nulables a objetos. Veámoslas:

- El **operador !**, conocido como *"assertion operator"* o *"non-null assertion operator"*, permite indicar al compilador de Dart que estamos seguros de que la variable que se accede no va a ser null, para que no genere error de compilación. Sin embargo, lo debemos usar con cuidado, ya que si el objeto fuese null en tiempo de ejecución se generará una excepción. En el ejemplo anterior su uso sería el siguiente:

  ```
  print(persona!.saludar());
  ```

- El **operador ?** se usa para acceder de manera segura a un campo de un objeto, cuando el objeto podría ser nulo. Este operador se llama *"operador de acceso seguro"* o *"nullable access operator"*, y su funcionamiento es el siguiente: si el objeto no es nulo, se accede a su propiedad correspondiente. Si el objeto es nulo, la expresión devuelve `null`, en lugar de generar una excepción. En nuestro ejemplo lo podemos combinar con el operador ?? del siguiente modo:

  ```
  print(persona?.saludar() ?? 'No hay persona');
  ```

6 Listas.

En Dart el concepto de *array* no existe como tal y, en su lugar, debemos trabajar con listas. Las listas son uno de los muchos tipos de colecciones que ofrece Dart. En concreto, una lista es una colección ordenada de elementos, donde cada elemento tiene un índice asociado que comienza en cero. Las listas son estructuras de datos dinámicas, lo que significa que pueden cambiar de tamaño durante la ejecución del programa, agregando o eliminando elementos según sea necesario.

Las listas están indexadas desde 0 en adelante; así, en una lista de tres elementos, el primero ocupará la posición 0, el segundo la 1 y el tercero la 2. Por ejemplo:

```
const lista = [3, -7, 6];       // Crea la lista [3, 1, 6];
print(lista[1]);                // Muestra -7 por la consola
lista[0] = 8;                   // Modifica el primer valor de la lista, asignándole un 8
```

6.1 Tipos de listas.

Atendiendo a su capacidad para ser modificadas, podemos crear tres tipos de listas en Dart:

- ***Listas con longitud dinámica***. Se trata de listas que pueden crecer. Se crean con los literales [], que opcionalmente podemos prefijar con <> para indicar el tipo de la lista. La lista anteriormente creada es un ejemplo de este tipo de lista. A continuación tenemos otro:

  ```
  const listaEnteros = const [1, 2, 3, 4];           // Crea una lista de tipo List<int>
  const listaNumeros = const <num>[1, 2.5, -3.25];   // Crea una lista de tipo List<num>

  listaEnteros[2] = 0;                  // Modifica el último elemento de la lista,
                                        // cambiando su valor por un 0
  listaEnteros[3] = 8;                  // No podemos añadir un cuarto valor de este modo,
                                        // Se generará una excepción
  listaEnteros.add(8);                  // Añade un cuarto valor a la lista correctamente
  ```

- ***Listas con longitud fija***. Su tamaño se determina cuando se crea la lista, y no se puede modificar posteriormente. En caso de intentarlo se generará una excepción. Ejemplo:

  ```
  var listaEnteros = List<int>.filled(3, 0);   // Crea la lista de longitud fija [0, 0, 0]
  listaEnteros.add(1);                         // Genera una excepción, ya que la lista
                                               // tiene longitud fija y no puede crecer
  ```

  Otras formas de crear listas de longitud fija son las siguientes:

  ```
  var listaEnteros1 = List<int>.generate(3, (i) => i); // Genera la lista [0, 1, 2]
  var listaVacia = List.empty(); // Genera una lista vacía de longitud fija
  ```

  Ambos constructores, `filled()` y `generated()`, admiten un parámetro `growable` que por defecto es `false`. De ahí que estas listas no puedan crecer, pero podríamos establecerlo a `true` en caso de que necesitemos que las listas puedan crecer:

  ```
  var listaQuePuedeCrecer = List<int>.generate(3, (i) => i, growable: true);
  ```

- ***Listas de solo lectura***. Dada una lista de longitud fija, podemos obtener una copia no modificable a partir de ella usando el método `unmodifiable()`. Ejemplo:

```
var listaInt = List<int>.generate(5, (index) => index);
var listaIntReadonly = List<int>.unmodifiable(listaInt);
listaIntReadonly[0] = 5;        // Excepción! La lista no se puede modificar
```

6.2 Iteración de listas.

Podemos iterar listas de forma muy cómoda con un bucle `for`. Por ejemplo:

```
var nombres = ['Alice', 'Bob', 'Charlie'];
for (var nombre in nombres) {
  print('Hola, $nombre');
}
```

En este caso, el bucle `for` itera sobre los elementos de la lista nombres y en cada iteración, la variable nombre toma el valor de uno de los elementos de la lista. El bloque de código dentro del bucle imprimirá `'Hola, '` seguido del nombre correspondiente. La salida será la siguiente:

```
Hola, Alice
Hola, Bob
Hola, Charlie
```

6.3 Manipulación de listas.

6.3.1 El operador de propagación.

El ***operador de propagación*** (…), *spread operator* en inglés, puede utilizarse con listas para copiar sus elementos en otra lista. Esto proporciona una forma concisa y conveniente de agregar los elementos de una lista a otra, ya sea al principio, al final o en cualquier posición específica. El operador de propagación se puede usar tanto con listas de tamaño fijo como con listas de tamaño dinámico. Ejemplo:

```
List<int> lista1 = [1, 2, 3];
List<int> lista2 = [4, 5, ...lista1, 6, 7]; // lista2 es [4, 5, 1, 2, 3, 6, 7]
```

Otro ejemplo:

```
List<int> lista3 = [0, ...lista1, 8, 9, ...[10, 11]]; // lista3 es [0, 1, 2, 3, 8, 9, 10, 11]
```

6.3.2 Los constructores *List.of()* y *List.from()*.

Los constructores `List.of()` y `List.from()` se utiliza para crear una nueva lista a partir de una colección existente o un iterable. Esto es útil cuando deseas copiar los elementos de otra lista o generar una nueva lista basada en datos de una fuente diferente. Observa que las copias que se hacen son poco profundas.

Ambos constructores son, por tanto, muy similares. Únicamente se diferencian en que `List.of()` no hace casting de tipos, mientras que `List.from()` si.

Ejemplo de uso de `List.of()`:

```dart
void main() {
  List<int> listaOriginal = [1, 2, 3, 4, 5];

  // Dado que el tipo de los elementos de listaOriginal coincide con el tipo de la lista destino,
  // podemos usar List.of sin problemas.
  List<int> nuevaLista = List<int>.of(listaOriginal);
}
```

Ejemplo de uso de `List.from()`:

```dart
void main() {
  List<Object> frutas = ['manzana', 'plátano', 'naranja'];

  // El constructor from realizará un casting de los elementos de frutas a String en tiempo de
  // ejecución. Si hubiésemos usado List.of, habríamos tenido un error de compilación
  List<String> nuevaLista = List<String>.from(frutas);
}
```

6.3.3 Métodos básicos para la manipulación de colecciones.

1. `add()`: Este método se utiliza para agregar un elemento al final de una lista. Por ejemplo:

```dart
List<int> lista = [1, 2, 3];
lista.add(4);
// Ahora 'lista' contiene [1, 2, 3, 4]
```

2. `addAll()`: Se usa para agregar todos los elementos de una colección iterable al final de la lista. Por ejemplo:

```dart
List<int> lista = [1, 2, 3];
lista.addAll([4, 5]);
// Ahora 'lista' contiene [1, 2, 3, 4, 5]
```

3. `insert()`: Este método se utiliza para insertar un elemento en una posición específica de la lista. Puedes especificar el índice en el que deseas insertar el elemento. Por ejemplo:

```dart
List<int> lista = [1, 2, 3];
lista.insert(1, 4);
// Ahora 'lista' contiene [1, 4, 2, 3]
```

4. `insertAll()`: Similar a addAll(), pero en lugar de agregar al final, puedes insertar una colección iterable en una posición específica de la lista.
5. `remove()`: Se utiliza para eliminar la primera ocurrencia de un elemento específico de la lista. Por ejemplo:

```dart
List<int> lista = [1, 2, 3, 4, 5];
lista.remove(3);
// Ahora 'lista' contiene [1, 2, 4, 5]
```

6. `removeAt()`: Elimina el elemento en el índice especificado de la lista.
7. `removeLast()`: Elimina el último elemento de la lista.

8. `shuffle()`: Mezcla los elementos de la lista en un orden aleatorio.
9. `contains()`: Verifica si la lista contiene un elemento específico y devuelve un valor booleano.

```
List<int> numbers = [1, 2, 3, 4, 5];
bool contains5 = numbers.contains(5); // Resultado: true
bool contains6 = numbers.contains(6); // Resultado: false
```

10. `indexOf()`: Devuelve el índice de la primera ocurrencia de un elemento específico en la lista. Si el elemento no está presente, devuelve -1.
11. `join()`: Convierte los elementos de la lista en una cadena única, concatenándolos con un separador específico. Por ejemplo:

```
List<String> fruits = ['Apple', 'Banana', 'Orange', 'Grapes'];
String joinedFruits = fruits.join(', ')
  .toList(); // Resultado: 'Apple, Banana, Orange, Grapes'
```

6.3.4 Métodos de primer orden para la manipulación de colecciones.

Los métodos de primer orden permiten manipular listas de forma muy concisa y flexible, dando lugar a código más legible y de mayor calidad. Estos métodos tienen su origen en el paradigma de programación funcional, de la familia de la programación declarativa. Su uso reduce la necesidad de usar bucles para procesar elementos en una lista y da lugar a un código más breve, legible y fácil de mantener. Algunos de los métodos más comunes son los siguientes:

1. `map()`: Este método se utiliza para transformar cada elemento de una lista aplicando una función a cada uno de ellos. Retorna una nueva lista con los resultados de la transformación. Ejemplo:

```
List<int> numbers = [1, 2, 3];
List<int> doubledNumbers = numbers.map((num) => num * 2).toList(); // Resultado: [2, 4, 6]
```

2. `where()`: Filtra los elementos de la lista que cumplen con una condición dada en una función de prueba y devuelve una nueva lista con los elementos que cumplen dicha condición. Ejemplo:

```
List<int> numbers = [1, 2, 3, 4, 5];
List<int> evenNumbers = numbers.where((num) => num % 2 == 0).toList(); // Resultado: [2, 4]
```

3. `sort()`: Ordena los elementos de la lista según un criterio específico. También puede recibir una función de comparación personalizada para ordenar elementos más complejos. Ejemplo:

```
List<int> numbers = [3, 1, 4, 2, 5];
numbers.sort().toList(); // Resultado: [1, 2, 3, 4, 5]
```

4. `toList()`, `toSet()`: Estos métodos convierten los datos resultantes de las operaciones anteriores en un nuevo objeto List o Set, respectivamente. Debemos usarlos después de aplicar otros métodos para obtener el resultado final como una lista o conjunto.
5. `any()`: Comprueba si algún elemento en la lista satisface una condición dada por una función de prueba. Retorna un valor booleano. Ejemplo:

```dart
List<int> numbers = [1, 2, 3, 4, 5];
bool hasEvenNumber = numbers.any((num) => num % 2 == 0); // Resultado: true
```

6. `every()`: Comprueba si todos los elementos de la lista satisfacen una condición específica dada por una función de prueba. Retorna un valor booleano. Ejemplo:

```dart
List<int> numbers = [2, 4, 6, 8, 10];
bool allEvenNumbers = numbers.every((num) => num % 2 == 0);  // Resultado: true
```

7. `reduce()`: Combina todos los elementos de una lista aplicando una función que toma dos elementos y retorna un solo valor. Este método usa el primer elemento de la lista como valor inicial, por lo que no se puede utilizar con una lista vacía. Además, hay que tener en cuenta que el tipo devuelto por este método siempre coincidirá con el tipo de los elementos de la lista sobre la que se aplica. Ejemplo:

```dart
List<int> numbers = [1, 2, 3, 4];
int sum = numbers.reduce((value, element) => value + element); // Resultado: 10
```

8. `fold()`: Similar al anterior, solo que permite especificar un valor inicial a partir del cual se va generando el resultado, aplicando la función que se indique. Por ello, este método puede aplicarse a listas vacías y devolver un tipo de dato que no tiene por qué coincidir con el de los elementos de la lista. Ejemplo:

```dart
List<string> source = <string>['x', 'yy', 'zzz'];
final length = source
      .fold<int>(0, (int currentCount, String element) => currCount + element.length);
// Resultado: 6 (suma de las longitudes de todos los strings del array)
```

A continuación podemos ver algunos ejemplos donde se encadenan varios de los métodos anteriores para realizar operaciones complejas sobre una lista. Supongamos que tenemos una lista de números y queremos filtrar los números pares, elevar cada número resultante al cuadrado y luego sumarlos todos. Podemos hacerlo encadenando los métodos `where()`, `map()` y `reduce()` de la siguiente manera:

```dart
void main() {
  List<int> numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

  int sumOfSquaresOfEvenNumbers = numbers
      .where((n) => n % 2 == 0) // Filtrar números pares
      .map((n) => n * n) // Elevar al cuadrado
      .fold(0, (value, element) => value + element); // Sumar

  print(
      sumOfSquaresOfEvenNumbers); // Resultado: 120 (2^2 + 4^2 + 6^2 + 8^2 + 10^2 = 120)
}
```

En este ejemplo, primero usamos el método `where()` para filtrar los números pares de la lista. Luego, utilizamos el método `map()` para elevar cada número al cuadrado. Finalmente, encadenamos el método `reduce()` para sumar todos los elementos resultantes y obtener el resultado final, que es la suma de los cuadrados de los números pares en la lista.

6.3.5 El operador de cascada.

El operador de cascada (`..`) se utiliza para realizar un encadenamiento de operaciones sobre un objeto. Esto significa que puedes realizar múltiples operaciones en el mismo objeto sin tener que repetir el nombre del objeto en cada línea. El operador `..` se usa para encadenar llamadas a métodos o propiedades que **no** devuelven el mismo objeto modificado, lo que permite encadenar varias operaciones de manera más concisa. A continuación tienes un ejemplo de uso del operador `..` en Dart:

```dart
class Persona {
  String nombre = '';
  int edad = 0;

  void establecerNombre(String nuevoNombre) {
    nombre = nuevoNombre;
  }

  void establecerEdad(int nuevaEdad) {
    edad = nuevaEdad;
  }
}

void main() {
  var persona = Persona()
    ..establecerNombre("Juan")
    ..establecerEdad(30);

  print("Nombre: ${persona.nombre}, Edad: ${persona.edad}");
}
```

En este ejemplo, hemos creado una clase Persona con propiedades nombre y edad, así como métodos `establecerNombre` y `establecerEdad`. Ambos métodos modifican el estado interno del objeto, pero no devuelven nada en absoluto. Sin embargo, al usar el operador `..` podemos encadenar la llamada a estos métodos en cualquier instancia de `Persona` (`persona` en este caso), lo que hace que el código sea más conciso y legible.

El resultado de este código será:

```
Nombre: Juan, Edad: 30
```

Su uso con listas resulta particularmente útil cuando queremos encadenar llamadas con métodos como `add()`, cuyo tipo de dato de salida es `void`. Por ejemplo:

```dart
void main() {
  List<int> numeros = [1, 2, 3, 4, 5];

  // Usando el operador en cascada para realizar múltiples operaciones en 'numeros'
  numeros
    ..add(6)
    ..addAll([7, 8, 9])
    ..remove(3);

  print(numeros); // Salida: [1, 2, 4, 5, 6, 7, 8, 9]
}
```

En este ejemplo, primero creamos una lista llamada `numeros` con algunos valores iniciales. Luego, usamos el operador en cascada (`..`) para realizar varias operaciones en la misma lista:

- `add(6)` agrega el número 6 al final de la lista.
- `addAll([7, 8, 9])` agrega los números 7, 8 y 9 al final de la lista.
- `remove(3)` elimina la primera ocurrencia del número 3 en la lista.

Como puedes ver, el operador en cascada nos permite encadenar múltiples operaciones en la misma lista de manera concisa y eficiente.

7 Diccionarios.

En Dart, los diccionarios son una estructura de datos conocida como "mapas" o "map". Representan una colección de pares clave-valor donde cada clave es única y se utiliza para acceder a su correspondiente valor asociado. En otras palabras, un diccionario es una colección de elementos en la que cada elemento tiene una clave y un valor relacionado. Ejemplo de diccionario:

```
Map<String, int> ages = {
  "Alice": 30,
  "Bob": 28,
  "Charlie": 35,
};
```

En este ejemplo, `ages` es un diccionario que mapea nombres (`String`) con sus edades (`int`). Cada `nombre` es una clave única que está asociada con una `edad` específica.

7.1 Uso de if y for en literales diccionario.

En Dart, es posible usar sentencias `if` y `for` dentro de un literal de diccionario (*map literal*) para construir diccionarios de forma más flexible y dinámica. Esto permite agregar elementos al diccionario de manera condicional o realizar transformaciones en los valores durante la creación del diccionario.

- Puedes usar `if` para agregar elementos al diccionario solo cuando se cumpla una condición específica. Es importante destacar que la condición debe ser evaluada como un valor booleano (`true` o `false`).
- Puedes utilizar `for` para iterar sobre un iterable (por ejemplo, una lista) y agregar elementos al diccionario en función de los elementos del iterable.

Ejemplo:

```
void main() {
  List<String> fruits = ['apple', 'banana', 'orange', 'grapes', 'mango'];

  // Crear un diccionario con los nombres de frutas que tienen más de 5 letras
  var fruitsDictionary = {
    for (var fruit in fruits)
      if (fruit.length > 5) fruit: fruit.length,
  };

  print(fruitsDictionary); // Resultado: {banana: 6, orange: 6, grapes: 6}
}
```

En este ejemplo, utilizamos un literal de diccionario para crear `fruitsDictionary`. Usamos la sentencia for para iterar sobre la lista `fruits` y agregar solo las frutas que tienen más de 5 letras como clave y la longitud de cada fruta como valor.

A partir de Dart 2.3, también es posible usar el operador `if` en el lado derecho de un `for` para filtrar elementos del iterable. Esto permite agregar más lógica condicional dentro del literal del diccionario. Ejemplo:

```
void main() {
  List<String> fruits = ['apple', 'banana', 'orange', 'grapes', 'mango'];
```

```
  // Crear un diccionario con los nombres de frutas que tienen más de 5 letras y la palabra
"fruit" en el nombre
  var fruitsDictionary = {
    for (var fruit in fruits if fruit.length > 5) '$fruit-fruit': fruit.length,
  };

  print(fruitsDictionary); // Resultado: {banana-fruit: 6, orange-fruit: 6, grapes-fruit: 6}
}
```

En este ejemplo, utilizamos el operador `if` en el lado derecho del `for` para agregar solo las frutas que tienen más de 5 letras y contienen la palabra *"fruit"* en su nombre al diccionario.

7.2 Diccionarios y el formato JSON.

7.2.1 Representación de estructuras JSON en Dart.

Los diccionarios son muy útiles para representar estructuras de datos complejas, como objetos JSON, ya que los objetos JSON también siguen el formato clave-valor. Veamos un ejemplo de estructura tipo JSON expresada mediante literales List y Map en Dart. Partimos del siguiente documento JSON:

```
{
  "nombre": "Ejemplo JSON",
  "descripcion": "Este es un ejemplo de un documento JSON",
  "fecha_creacion": "2023-10-10",
  "activo": true,
  "valores": [1, 2, 3, 4, 5],
  "usuario": {
    "nombre": "UsuarioEjemplo",
    "email": "usuario@example.com"
  }
}
```

Podemos expresar esta estructura de datos en Dart mediante List y Map, y asignárla a una variable. Quedaría del siguiente modo:

```
void main() {
  Map<String, dynamic> ejemploJson = {
    "nombre": "Ejemplo JSON",
    "descripcion": "Este es un ejemplo de un documento JSON",
    "fecha_creacion": "2023-10-10",
    "activo": true,
    "valores": <int>[1, 2, 3, 4, 5],
    "usuario": <String, dynamic>{
      "nombre": "UsuarioEjemplo",
      "email": "usuario@example.com"
    }
  };

  print(ejemploJson);
}
```

Existe la posibilidad de omitir la declaración de tipo delante de los literales List y Map, pero su uso es recomendable ya que permite aprovechar las ventajas del sistema de tipos de Dart.

7.2.2 Procesamiento de cadenas JSON en Dart.

Lo normal es leer datos JSON desde un archivo o desde una API REST. En ambos casos, partimos de una cadena de caracteres donde se halla representado el JSON, y es necesario su análisis y procesamiento para transformarlo en una estructura de datos basada en List y Map, similar a la vista en el apartado anterior.

Para representar objetos JSON utilizando diccionarios en Dart, puedes hacer lo siguiente:

```
void main() {
  String jsonString = '{"name": "John", "age": 25, "email": "john@example.com"}';

  // Convertimos una cadena JSON en un diccionario
  Map<String, dynamic> person = jsonDecode(jsonString);

  // Accedemos a los campos del diccionario
  String name = person['name'];
  int age = person['age'];
  String email = person['email'];

  // Imprimimos los datos
  print('Name: $name');
  print('Age: $age');
  print('Email: $email');
}
```

En este ejemplo, utilizamos la función `jsonDecode()` para convertir el JSON en una estructura de datos de Dart (un diccionario). Luego, podemos acceder a los campos del diccionario utilizando las claves correspondientes. La clave-valor `"name": "John"` se convierte en el par `person['name'] = "John"` y así sucesivamente.

7.2.3 Lectura de datos JSON desde un fichero.

Si queremos leer los datos de en un fichero y hacer uso de ellos en nuestra aplicación, podemos hacerlo mediante la clase `File`. Esta clase se utiliza para leer, escribir o manipular archivos en tu dispositivo. Para trabajar con ella primero debes importar la biblioteca `dart:io`.

A continuación puedes ver un ejemplo de cómo usar `File` para leer un archivo JSON, que en este caso contiene una lista de objetos, y su posterior conversión en una `List<Map<String, dynamic>>`. Supongamos que tienes un archivo llamado *data.json* en el directorio raiz de tu proyecto Dart con el siguiente contenido:

```
[
  { "nombre": "Manolo", "edad": 25 },
  { "nombre": "Raquel", "edad": 30 },
  { "nombre": "Laura", "edad": 35 }
]
```

El código necesario para leer y procesar este archivo sería el siguiente:

```
import 'dart:io';
import 'dart:convert';

void main() async {
  // Ruta al archivo JSON
  final filePath = 'data.json';

  try {
    // Crear una instancia de File para el archivo JSON
```

```
    final file = File(filePath);

    // Leer el contenido del archivo como una cadena
    final jsonString = await file.readAsString();

    // Decodificar la cadena JSON en una lista de mapas
    final List<Map<String, dynamic>> listaDePersonas = jsonDecode(jsonString);

    // Imprimir la lista de objetos
    for (var persona in listaDePersonas) {
      print('Nombre: ${persona['nombre']}, Edad: ${persona['edad']}');
    }
  } catch (e) {
    print('Error al leer el archivo: $e');
  }
}
```

En este ejemplo:

1. Importamos las bibliotecas necesarias: `dart:io` para trabajar con archivos y `dart:convert` para decodificar el JSON.
2. Declaramos la función `main()` como `async`, ya que dentro de ella vamos a usar un método asíncrono con `await` (por eso `main()` tiene la palabra `async` al final).
3. Especificamos la ruta al archivo JSON que queremos leer (*data.json* en este caso).
4. Usamos `File` para crear una instancia que representa el archivo.
5. Usamos `await file.readAsString()` para leer el contenido del archivo como una cadena. El uso de `await` se debe a que `readAsString()` es un método asíncrono. Para poder usar `await`, la función donde se usa debe estar declarada como `async`.
6. Utilizamos `jsonDecode()` para decodificar la cadena JSON en una lista de mapas.
7. Finalmente, recorremos la lista de mapas e imprimimos los valores de `"nombre"` y `"edad"` de cada objeto en la lista.