DART

DISEÑO DE INTERFACES WEB

ÍNDICE

INTRODUCCIÓN A DART **3**

¿QUÉ ES DART? 3
¿QUÉ HACE A DART ESPECIAL? 3
¿DÓNDE PUEDO UTILIZAR DART? 3

INSTALACIONES NECESARIAS **5**

SDK DE FLUTTER EN WINDOWS 5
SDK DE FLUTTER EN MACOS 6

CREACIÓN DE UN PROYECTO DE DART CON INTELLIJ **8**

PRIMEROS PASOS EN DART 9

SINTAXIS DART **10**

UN PROGRAMA BÁSICO DE DART 10
COMENTARIOS *10*
CONCEPTOS IMPORTANTES 11
DART NULL SAFETY: TIPOS QUE NO ACEPTAN VALORES NULOS 12
SISTEMA DE TIPO DART *12*
DECLARACIÓN DE VARIABLES QUE NO ACEPTAN VALORES NULL *13*
DECLARACIÓN DE VARIABLES QUE ACEPTAN VALORES NULL *14*
VARIABLES 15
VALOR PREDETERMINADO *15*
FINAL Y CONSTANTE *16*
TIPOS INTEGRADOS 17
NÚMEROS *17*
STRING *19*
BOOLEANOS *20*
LISTAS *20*
MAPS *21*
FUNCIONES 22
CONTROL DE FLUJO 24
CLASES 25
MIXINS *27*
SOPORTE ASÍNCRONO 30
FUTURES *30*
ASYNC AWAIT *31*

INTRODUCCIÓN A DART

¿QUÉ ES DART?

Dart es un lenguaje Open Source desarrollado en Google con el objetivo de permitir a los desarrolladores utilizar un lenguaje orientado a objetos y con análisis estático de tipo. Se dice de un lenguaje de programación que usa un tipado estático cuando la comprobación de tipificación se realiza durante la compilación, y no durante la ejecución. Desde la primera versión estable en 2011, Dart ha cambiado bastante, tanto en el lenguaje en sí como en sus objetivos principales. Con la versión 2.0, el sistema de tipo de Dart pasó de opcional a estático, y desde su llegada, Flutter se ha convertido en el principal objetivo del lenguaje.

¿QUÉ HACE A DART ESPECIAL?

A diferencia de muchos lenguajes, Dart se diseñó con el objetivo de hacer el proceso de desarrollo lo más cómodo y rápido posible para los desarrolladores. Por eso, viene con un conjunto bastante extenso de herramientas integradas, como su propio gestor de paquetes, varios compiladores, un analizador y formateador. Además, la máquina virtual de Dart y la compilación *Just-in-Time* (forma de ejecutar el código informático que implica la compilación durante la ejecución de un programa, en tiempo de ejecución, y no antes de la ejecución) hacen que los cambios realizados en el código se puedan ejecutar inmediatamente.

Una vez en producción, el código se puede compilar en lenguaje nativo, por lo que no es necesario un entorno especial para ejecutarlo. En caso de que se haga desarrollo web, Dart se transpila a JavaScript (se pasa a JavaScript y luego se compila).

En cuanto a la sintaxis, la de Dart es muy similar a lenguajes como JavaScript, Java y C ++, por lo que aprender Dart sabiendo uno de estos lenguajes es cuestión de horas.

Además, Dart consta de un gran apoyo para la asincronía, y trabajar con generadores e iterables es extremadamente sencillo.

¿DÓNDE PUEDO UTILIZAR DART?

Dart es un lenguaje de propósito general, y lo puedes utilizar casi para cualquier cosa:

- En aplicaciones web, utilizando la librería de arte: HTML y el transpilador para transformar el código en Dart en JavaScript, o utilizando Frameworks como AngularDart.

- En servidores, utilizando las librerías de arte: HTTP y arte: IO. También hay varios Frameworks que se pueden utilizar, como por ejemplo Aqueduct.
- En aplicaciones de consola.
- En aplicaciones móviles gracias a Flutter.

INSTALACIONES NECESARIAS

Durante la introducción a Dart utilizaremos IntelliJ Idea Community, además necesitaremos instalar el SDK de DART, en nuestro caso como el curso está orientado a Flutter, instalaremos directamente este último que ya contiene el SDK de Dart.

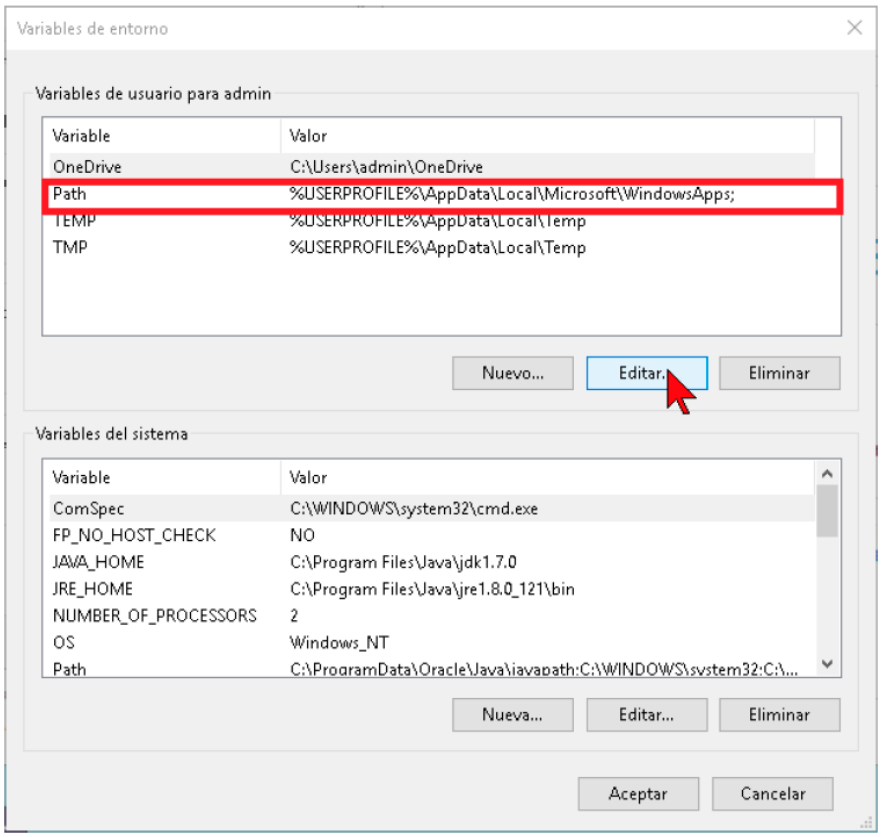
https://flutter.dev/docs/get-started/install

SDK DE FLUTTER EN WINDOWS

Desde el enlace anterior pinchamos en **Windows** y descargamos el .zip con el SDK, descomprimimos el contenido en la raíz de nuestro disco duro en una carpeta que llamaremos "**Flutter**"', una vez haya terminado de la descompresión buscamos en la barra de Windows "*variables de entorno*", seleccionamos "*Path*" en el recuadro superior y presionamos **Editar**. Presionamos **Nuevo** e introducimos la ruta donde descomprimimos anteriormente el SDK de Flutter añadiendo a la carpeta **bin** al final:

C:\Flutter\bin

Para comprobar si se ha instalado correctamente, abrimos una nueva consola de comandos y ejecutamos el comando ***Flutter --version***. Si todo ha ido bien debería aparecernos la versión actual de Flutter.



SDK DE FLUTTER EN MACOS

Desde el enlace de arriba vamos a la sección para **Mac** y descargamos el archivo .zip del SDK, lo descomprimimos. Dentro de nuestra carpeta **Home** o carpeta de usuarios, creamos una carpeta llamada "**development**" y dentro de ella copiamos la carpeta **Flutter** que se ha descomprimido.

Ahora vamos a ejecutar una serie de comandos desde la consola para exportar Flutter a nuestras variables de entorno.

Abrimos la terminal y escribimos "***pwd***" para asegurarnos que estemos en la carpeta **Home**. Nos debería aparecer algo similar a esto:

/User/NombreDeUsuario.

Si no fuera así debemos navegar con cd hasta dicha ruta.

Una vez estamos en la ruta debemos configurar en el archivo ***bash_profile***, este archivo puede existir o no así que para asegurarnos ejecutamos:

vim .bash_ profile

Esto nos abrirá el editor **Vim**. Seguimos los siguientes pasos:

1. Presionamos la tecla "i" del teclado para empezar a insertar texto
2. Copiamos la línea que aparece en la página de Flutter:

   export PATH="$PATH:`pwd`/flutter/bin"
3. Eliminamos todo lo que va después de $PATH:
4. Abrimos el Finder y navegamos hasta la carpeta ***bin*** de que está dentro de la carpeta ***development/flutter/***
5. Arrastramos la carpeta ***bin*** al editor de Vim justo delante de $PATH: y cerramos la línea con dobles comillas con cuidado de no dejar espacios extra. Debe quedar de la siguiente manera:

   export PATH="$PATH:/Users/ YOUR_FLUTTER_DIR /development/flutter/bin".

   Donde *YOUR_FLUTTER_DIR* es el nombre de usuario del sistema.
6. Para guardar y salir presionamos la tecla **ESC** y escribimos ":wq!" (debe aparecer abajo del todo) y presionamos Enter.
7. Cerramos nuestra terminal del todo y volvemos a abrir una, si escribimos *Flutter*

   --version comprobaremos si todo ha ido bien.

Si el terminal que usa zsh mostrará el siguiente error:

zsh: command not found: flutter

Para solucionarlo:

1. Abre un terminal
2. Escribe: *vim $HOME/.zshrc*
3. Presiona la tecla “I” para entrar en modo INSERT
4. Añade la siguiente línea:

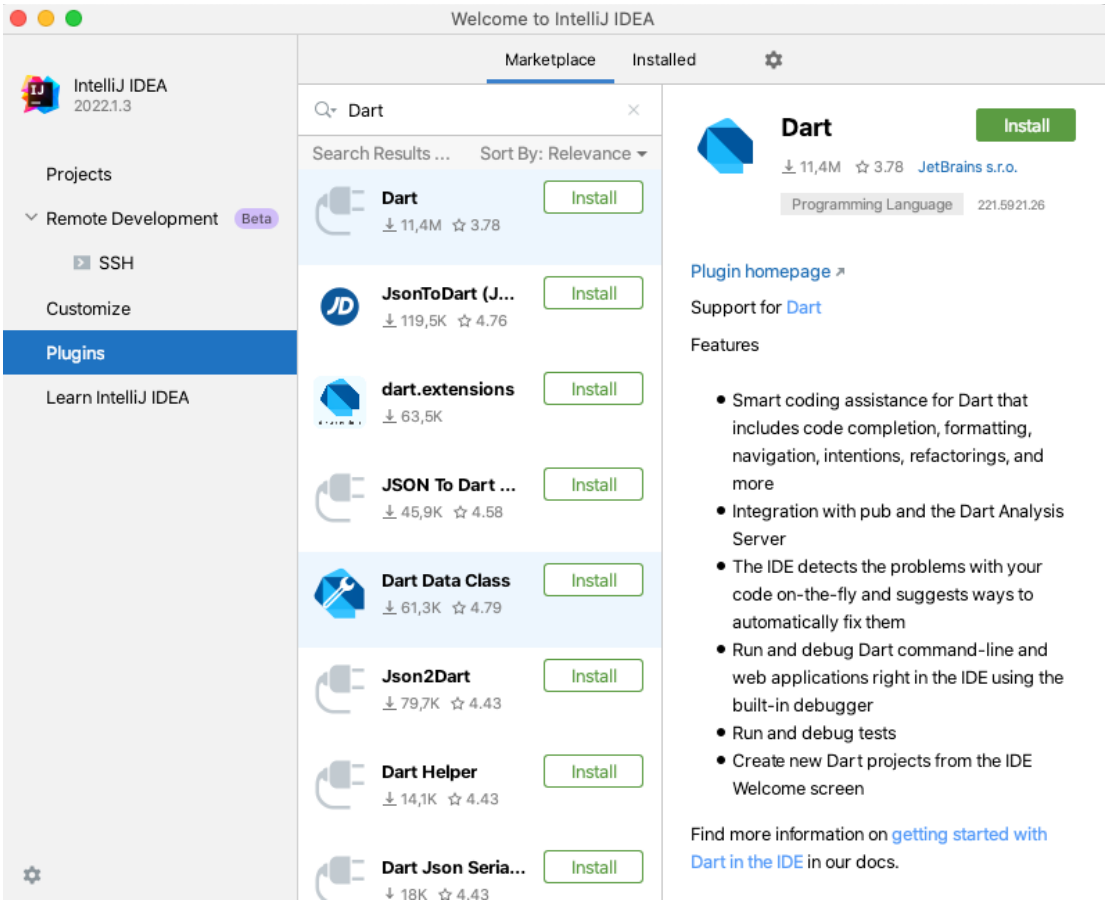
export PATH="$PATH:/YOUR_FLUTTER_DIR/flutter/bin"

5. Presiona la tecla **ESC** y escribe *:wq!* en el terminal y presiona la tecla **Enter** para salir de Vim.
6. Reinicia el terminal y escibe “*flutter doctor*”

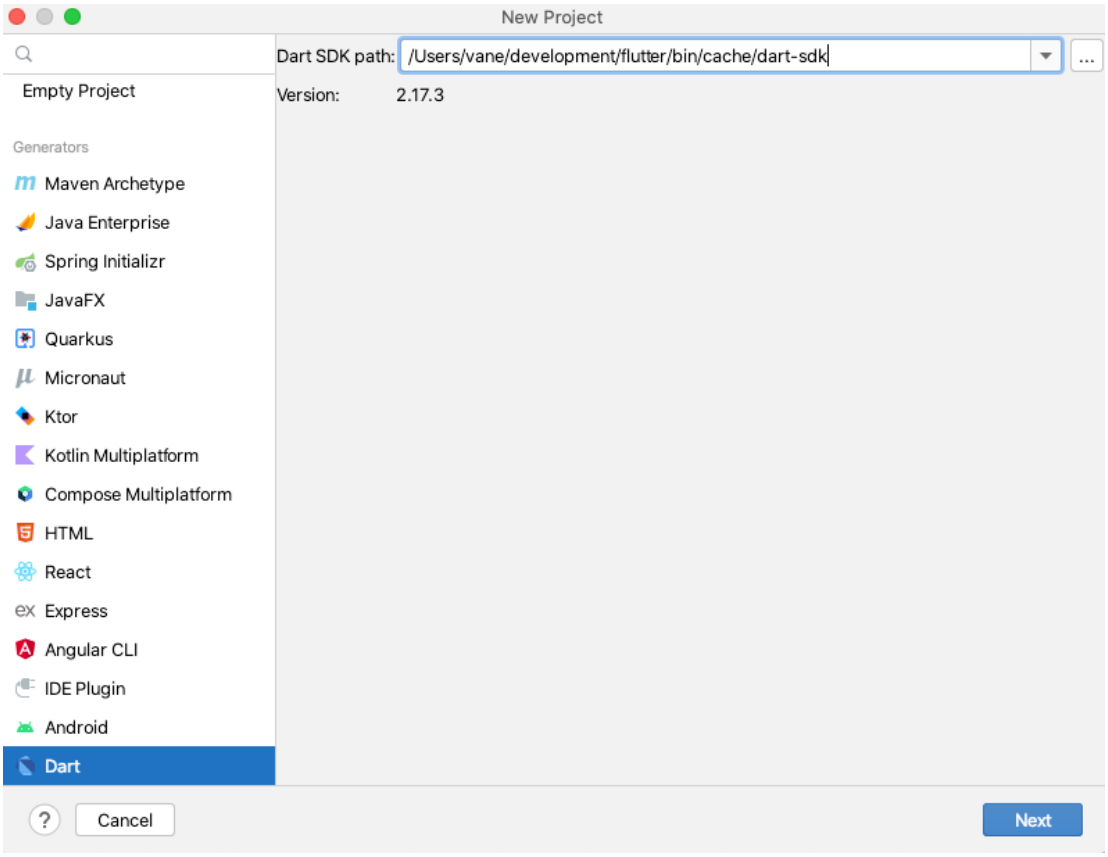
CREACIÓN DE UN PROYECTO DE DART CON INTELLIJ

Abrimos el programa y antes de hacer nada debemos instalar la extensión de Dart.

En el menú lateral presionamos en *Plugins* y buscamos **Dart**, instalamos el oficial de JetBrains:



Una vez instalado vamos a *Projects* y seleccionamos *New Project*, en el menú de la izquierda seleccionamos *Dart*, nos pedirá el SDK de Dart, este se encuentra en la carpeta donde tenemos el SDK de Flutter dentro de **bin/cache/dart-sdk**. Si hemos seleccionado bien la ruta debería aparecernos abajo la versión de Dart. Presionamos siguiente, escogemos nombre y ruta del proyecto.



PRIMEROS PASOS EN DART

Para crear un archivo desde la carpeta raíz pulsamos el botón derecho y creamos un nuevo archivo (*New File*) cuya extensión tiene que ser *".dart"*

En Dart como en otros lenguajes, el programa siempre empieza a ejecutarse desde el método **main**. Por lo tanto, debemos tener esto muy en cuenta.

Lo que está antes del nombre de la función **main** es el tipo de variable de retorno, lo que va a devolver la función, en el caso del método **main** devolverá *void*. Si no especificamos nada, Dart entiende que vamos a devolver algo de tipo *dynamic*, es decir, algo de cualquier tipo.

Para ejecutar nos ponemos sobre el método **main** y pulsamos el botón derecho para que salga el menú contextual y elegimos la opción "**run**".

SINTAXIS DART

UN PROGRAMA BÁSICO DE DART

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

```
// Definir una función.
void printInteger(int aNumber) {
        print('El número es $aNumber.') ; // Imprimir en la consola.
}

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

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

COMENTARIOS

```
// This is a comment.
```

Un comentario de una sola línea. Dart también admite comentarios multilínea y de documentos.

Los comentarios en Dart/Flutter son los mismo que en otros lenguajes:

```
// Para comentarios de una línea

/*
 *Para comentarios
 * de varias líneas
*/
```

```
void
```

Un tipo especial que indica que nunca se devuelve nada.

```
int
```

Otro tipo que indica que es un número entero. Otros tipos son String, List y bool

```
42
```

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

```
print()
```

Una forma práctica de mostrar la salida. '...'(o "...") Un literal de cadena.

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

Interpolación de cadenas: incluye el equivalente de cadena de una variable o expresión dentro de un literal de cadena.

```
main()
```

La función especial, obligatoria y de nivel superior donde comienza la ejecución de la aplicación.

```
var
```

Una forma de declarar una variable sin especificar su tipo. El tipo de esta variable (int) está determinado por su valor inicial (42).

CONCEPTOS IMPORTANTES

- Todo lo que se puede colocar en una variable es un objeto, y cada objeto es una instancia de una clase. Los números pares, las funciones y el valor nulo son objetos. Con la excepción de nulo, todos los objetos heredan de la clase Object.
- Aunque Dart está fuertemente tipado, las anotaciones de tipo son opcionales porque Dart puede inferir tipos. En el código anterior, se infiere que el *number* es de tipo *int*.
- A diferencia de Java, Dart no tiene las palabras clave *public*, *protected* y *private*. Si un identificador comienza con un guion bajo (_), es privado para su biblioteca.
- Los identificadores pueden comenzar con una letra o guion bajo (_), seguido de cualquier combinación de esos caracteres más dígitos.

DART NULL SAFETY: TIPOS QUE NO ACEPTAN VALORES NULOS

SISTEMA DE TIPO DART

Se dice que Dart tiene un sistema de tipo **sound** (será porque a medida que escribes "escucha" tus cambios y te informa tus errores). Cuando escribimos código Dart, el verificador de tipos se asegura de que no podamos escribir algo como esto:

```
int age = "hello world"; // A value of type `String` can't be assigned to a
                         // variable of type `int`
```

Este código produce un error que nos dice que "*String no se puede asignar un valor a una variable de tipo int*".

De manera similar, cuando escribimos una función en Dart, podemos especificar un tipo de retorno. Debido a la seguridad del tipo, Dart puede garantizar con un 100% de confianza que esta función siempre devuelve un int. La seguridad de tipos nos ayuda a escribir programas más seguros y a razonar más fácilmente sobre el código. Pero la seguridad de tipos por sí sola no puede garantizar que una variable (o un valor de retorno) no sea ***null***.

Como resultado, este código se compila, pero genera una excepción en tiempo de ejecución:

```
square(null);
//Excepción no controlada: NoSuchMethodError: llamó al método '*' en null.
```

En este ejemplo, es bastante fácil detectar el problema. Pero en bases de código grandes es difícil hacer un seguimiento de lo que puede y no puede ser **null**. Las comprobaciones **null** en tiempo de ejecución pueden mitigar el problema, pero agregan más ruido:

```
int square(int value){
        assert(value != null); //for debugging
        if(value == null) throw Exception();
        return value * value;
}
```

Lo que realmente queremos aquí es decirle a Dart que el argumento *value* nunca debería ser **null**. Se necesita una mejor solución, y ahora la tenemos.

Dart presenta **Sound Null Safety** como una función de lenguaje y aporta tres beneficios principales:

- Podemos escribir código seguro para valores nulos con sólidas garantías de tiempo de compilación. Esto nos hace productivos porque Dart puede decirnos cuando estamos haciendo algo mal.
- Podemos declarar más fácilmente nuestra intención. Esto conduce a una API que se autodocumenta y es fácil de usar.
- El compilador de Dart puede optimizar nuestro código, lo que resulta en programas más pequeños y rápidos.

Así que veamos cómo funciona la seguridad nula en la práctica.

DECLARACIÓN DE VARIABLES QUE NO ACEPTAN VALORES NULL

El cambio de lenguaje principal es que de forma predeterminada todos los tipos ahora no admiten nulos.

Esto significa que este código no se compila:

```
void main(){
    int age; //valor no puede ser null
    age = null; //el valor es de tipo 'Null' y es asignado a un tipo 'int'
}
```

Al usar variables que no aceptan valores NULL, debemos seguir una regla importante:

"Las variables que no aceptan valores NULL siempre deben inicializarse con valores que no sean nulos."

Si razona en este sentido, será más fácil comprender todos los nuevos cambios de sintaxis. Repasemos este ejemplo:

```
int square(int value){
        return value * value;
}
```

Aquí se garantiza que tanto el argumento como el valor de retorno no son **null**. Como resultado, las comprobaciones **null** en tiempo de ejecución ya no son necesarias y este código ahora produce un error en tiempo de compilación:

```
square(null);
```

Pero si todos los tipos ahora no aceptan valores NULL de forma predeterminada, ¿cómo podemos declarar variables que aceptan valores NULL?

DECLARACIÓN DE VARIABLES QUE ACEPTAN VALORES NULL

El símbolo ? es lo que necesitamos:

```
String? name; //inicializa el valor a null
int? age = 36; //inicializa el valor no null
age = null; //se puede reasignar a null
```

Nota: no es necesario inicializar una variable que acepte valores NULL antes de usarla. Se inicializa null de forma predeterminada.

Las variables que aceptan valores NULL son una buena forma de expresar la ausencia de un valor y esto es útil en muchas API. Cuando diseñe una API, pregúntese si una variable debe ser **null** o no, y declare en consecuencia. Pero hay casos en los que sabemos que algo no puede ser **null**, pero no podemos demostrárselo al compilador. En estos casos, el operador de aserción puede ayudar.

Podemos usar el operador de aserción ! para asignar una expresión **null** a una variable no anulable:

```
int? maybeValue = 42;
int value = maybeValue!; //Válida, el valor no es null
```

Al hacer esto, le estamos diciendo a Dart que *maybeValue* no es **null**, y es seguro asignarlo a una variable que no acepta valores NULL.

Tenga en cuenta que la aplicación del operador de aserción a un **null** valor arrojará una excepción de tiempo de ejecución:

```
String? name;
print(name!); //valor nulo inesperado
print(null!); //valor nulo inesperado
```

Cuando sus suposiciones son incorrectas, el ! operador genera excepciones en tiempo de ejecución.

En resumen:

- Intente crear variables que no admitan valores NULL cuando sea posible, ya que se garantizará que no lo estén **null** en el momento de la compilación.
- Si sabe que una expresión anulable no será **null**, puede asignarla a una variable no anulable con el ! operador.

VARIABLES

He aquí un ejemplo de creación de una variable e inicialización:

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

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

Se infiere que el tipo de la variable de nombre es *String*, pero puede cambiar ese tipo especificándolo. Si un objeto no está restringido a un solo tipo, especifique el tipo de objeto (o dinámico si es necesario).

```
Object name = 'Bob';
```

Otra opción es declarar explícitamente el tipo que se inferiría:

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

VALOR PREDETERMINADO

Las variables no inicializadas que tienen un tipo *null* tienen un valor inicial de *null*. (Si no has optado por la seguridad nula, entonces cada variable tiene un tipo anulable). Incluso las variables con tipos numéricos son inicialmente nulas, porque los números, como todo lo demás en Dart, son objetos.

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

Si habilita la seguridad nula, debe inicializar los valores de las variables no nulas antes de utilizarlas:

```
int lineCount = 0;
```

No tiene que inicializar una variable local donde se declara, pero sí debe asignarle un valor antes de usarla. Por ejemplo, el siguiente código es válido porque Dart puede detectar que *lineCount* no es nulo en el momento en que se pasa a *print*():

```
int lineCount;

if (weLikeToCount) {lineCount = countLines();
} else {lineCount = 0;
}print(lineCount);
```

FINAL Y CONSTANTE

Si nunca tiene la intención de cambiar una variable, use ***final*** o ***const***, ya sea en lugar de *var* o en un tipo explicito. La diferencia entre **final** y ***const*** es que la constante ya está preestablecida en tiempo de compilación, mientras que **final** puede ser establecida durante la compilación. Por ejemplo, podemos recibir un array de String desde otra clase y asignarla como final en la clase receptora.

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

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

No se puede cambiar el valor de una variable final:

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

Cuando declare la variable, establezca el valor en una constante de tiempo de compilación, como un número o un literal de cadena, una variable *const* o el resultado de una operación aritmética en números constantes:

```
const bar = 1000000; // Unit of pressure (dynes/cm2)
const double atm = 1.01325 * bar; // Standard atmosphere
```

TIPOS INTEGRADOS

El lenguaje Dart tiene un soporte especial para lo siguiente:

- **Numbers (int, double)**
- **Strings (String)**
- **Booleans (bool)**
- **Lists (List, también conocidos como arrays)**
- Sets (Set)
- **Maps (Map)**
- Runes (Runes; con frecuencia reemplazados por caracteres en API)
- Symbols (Symbol)
- The value null (Null)

Algunos otros tipos también tienen funciones especiales en el lenguaje Dart:

- **Object: la superclase de todas las clases de Dart excepto Null.**
- Enum: La superclase de todas las enumeraciones.
- **Future** y Stream: se utiliza en el soporte de asíncrona.
- Iterable: se utiliza en bucles for-in y en funciones de generador síncrono
- Never: indica que una expresión nunca puede terminar de evaluarla con éxito. Se utiliza con mayor frecuencia para funciones que siempre lanzan una excepción.
- dynamic: Indica que desea deshabilitar la verificación estática. Por lo general, debe usar Object u Object? en cambio.
- void: indica que nunca se utiliza un valor. A menudo se utiliza como tipo de devolución.

NÚMEROS

Hay dos tipos de números:

- ***int***: Valores enteros no superiores a 64 bits, dependiendo de la plataforma. En las plataformas nativas, los valores pueden ser de -263 a 263 - 1. En la web, los valores enteros se representan como números JavaScript (valores de coma flotante de 64 bits sin parte fraccionada) y pueden ser de -253 a 253 - 1.
- ***double***: Números de coma flotante de 64 bits (doble precisión), según lo especificado por la norma IEEE 754.

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

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

Si un número incluye un decimal, es un doble. Estos son algunos ejemplos de definición de literales dobles:

```
var y = 1,1;
var exponents = 1,42e5;
```

También puedes declarar una variable como ***num***. Si lo haces, la variable puede tener valores enteros y dobles.

```
num x = 1; // x puede tener valores int y doblesx += 2,5;
```

Los literales enteros se convierten automáticamente en dobles cuando es necesario:

```
doble z = 1; // Equivalente a doble z = 1,0.
```

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

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

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

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

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

STRING

Una cadena Dart (objeto String) contiene una secuencia de unidades de código UTF-16. Puedes usar comillas simples o dobles para crear una cadena:

```
var s1 = ‘Las comillas simples funcionan bien para los literales de cadena’;
var s2 = "Las comillas dobles funcionan igual de bien";
var s3 = 'It\'s easy to escape the string delimiter.';
var s4 = "It's even easier to use the other delimiter.";
```

Puede poner el valor de una expresión dentro de una cadena usando ${expresión}. Si la expresión es un identificador, puede omitir {}. Para obtener la cadena correspondiente a un objeto, Dart llama al método *toString*() del objeto.

```
var s = 'string interpolation';

assert('Dart has $s, which is very handy.' ==
    'Dart has string interpolation, '
        'which is very handy.');
assert('That deserves all caps. '
        '${s.toUpperCase()} is very handy!' ==
    'That deserves all caps. '
        'STRING INTERPOLATION is very handy!');
```

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

Puedes concatenar cadenas usando literales de cadena adyacentes o el operador **+**:

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

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

BOOLEANOS

Para representar valores booleanos, Dart tiene un tipo llamado **bool**. Solo dos objetos tienen tipo **bool**: los literales booleanos verdadero y falso, que son constantes en tiempo de compilación.

La seguridad de tipo de Dart significa que no puede usar código como if (nonbooleanValue) o aserción (nonbooleanValue). En su lugar, verifica explícitamente los valores, como este:

```
// Comprueba si hay una cadena vacía.
var fullName = '';
assert(fullName.isEmpty);

// Comprueba si hay cero.
var hitPoints = 0;
assert(hitPoints <= 0);

// Comprueba si hay nulo.
var unicornio;
assert(unicornio == null);

// Comprueba si hay NaN.
var iMeantToDoThis = 0 / 0;
assert(iMeantToDoThis.isNaN);
```

LISTAS

Dart representa arrays en forma de objetos **List**. Una lista es simplemente un grupo ordenado de objetos. La biblioteca *dart:core* proporciona la clase **List** que permite la creación y manipulación de listas.

```
void main () {
        // Para declarar un List se hace de la siguiente manera
        // podemos comenzar una lista vacía
        List numeros = [1,'pepe'];

        // o de la siguiente manera, dando los valores iniciales
        List listaNumeros = [ 0,1,2,3,4,5,6,7,8,9,10];

        // Pero esto crea una lista de tipo dinámico de tal manera que lo
        // siguiente es válido
        listaNumeros.add('Pepe');//-> Añadir al final del List
        print(listaNumeros);

        // Para crear una lista de un único tipo:
        List<int> listadoNumeros = [0,1,2,3,4,5,6,7,8,9,10];
        print(listadoNumeros.length); // ->Imprime el tamaño del List
```

```
        //Para cambiar un valor
        listadoNumeros[0] = 30;

        // Generar un List. Utilizamos el método estático generate
        // Con esto le vamos a decir que genere 100 números,desde 0 hasta 100
        // y que lo guarde en la lista
        final numerosGenerados = List.generate(100, (int index) =>
        index);

        // Acceder a un dato del List
        print(numerosGenerados[1]);
}
```

MAPS

En general, un mapa es un objeto que asocia claves y valores. Tanto las claves como los valores pueden ser cualquier tipo de objeto. Cada clave se produce solo una vez, pero puedes usar el mismo valor varias veces.

Un mapa es una colección dinámica. En otras palabras, **Maps** puede crecer y reducirse en tiempo de ejecución.

Los mapas se pueden declarar de dos formas:

- Uso de literales de mapas
- Usando un constructor de mapas

```
void main(){
        //Definicion de un Map(podemos utilizar final, var, const ...)
        //Map<K,V>, le damos primero la clave y luego su valor, si no le
        //damos tipo es dinámico en ambos casos
        Map per = new Map();

        Map persona = {
                'nombre'        : 'Pepe',
                'edad'          : 35,
                'soltero'       : true,
                True            : false,
                1               : 200,
        };

        print( persona );

        // Acceder a un propiedad en concreto-> Le pasamos el key del valor
        // que queremos
        print(persona['nombre']);
        print(persona[true]);
        print(persona[1]);
```

```
        // En el caso anterior al no estar especificado el tipo de datos del
        // Map es <dynamic, dynamic> es recomendado al menos establecer el
        // tipo de datos de la clave
        Map<String, dynamic> persona2 = {
                'nombre'        : 'Pepe',
                'edad'          : 35,
                'soltero'       : true
        };

        //Añadir un par clave:valor directamente
        persona2.addAll({'3':400});

        //o creamos un mapa y luego lo añadimos
        Map<String, Dynamic> temp = {
                'altura': 1.78
        }
        persona2.addAll(temp);

        print(persona2);
}
```

FUNCIONES

Dart es un lenguaje de programación orientado a objetos de manera pura, esto significa que el propio lenguaje establece que las funciones tengan un tipo específico:

```
tipoADevolverPorLaFuncion NombreDeLaFuncion (ParametroQueRecibe){
}
```

Ejemplo:

```
// Las funciones se declaran prácticamente igual que en el resto de lenguajes
bool esPar(int numero){
        return numero % 2 == 0;
}

// NOTACIÓN ABREVIADA
/*
        No es necesario establecer el tipo de retorno, y en el caso de que
        solo contenga una línea como en la función anterior podemos expresar
        la función de la siguiente forma
*/
esImpar(int numero) => numero % 2 != 0;  // Se utiliza mucho en Flutter
```

```
void main(){
        // Para llamar a la función
        print(esPar(2));
        print(esImpar(3));
}
```

A la hora definir nuestras funciones, Dart nos da varias posibilidades en cuanto a los parámetros que esa función va a recibir.

- *Parámetros nombrados*: Nos permite darle un nombre concreto a los parámetros que recibe la función de tal manera que cuando llamemos a esa función, estemos obligados a pasarle el nombre del parámetro y el valor en el orden que nosotros queramos.
- *Parámetros opcionales*: Podemos establecer que los parámetros que reciba una función en Dart sean opcionales, e incluso en caso de no llegar ese parámetro, asignarle un valor por defecto.
- *Parámetros nombrados obligatorios*: Al utilizar parámetros nombrados, Dart los establece automáticamente como opcionales, pero podemos indicarle que son obligatorios con la palabra reservada **required**.

```
void main () {
        mostrarNombreYApellidos(apellidos: 'Garcia');
        mostrarNombreYApellidos2(apellidos: 'Garcia' , nombre: 'Manolo');
}

// PARAMETROS NOMBRADOS-> Para establecerlos utilizamos {}
// Esto nos permite llamar a la función pasándole los argumentos en el orden
// que queramos. Solo debemos especificar su nombre, es decir, el key:valor.
// PARAMETROS OPCIONALES->  Como Dart es nullSafety, Dart no interpreta que
// pueden ser nulos, por lo que debemos indicarle con ? o igualándolo a un //
valor por defecto
mostrarNombreYApellidos({ String? nombre, String apellidos = 'No name'}){
        print( '$nombre $apellidos');
}

// PARAMETROS NOMBRADOS OBLIGATORIOS-> También podemos especificar que aunque
// nuestros argumentos sean por nombre y no por posición necesitan un valor,
// es decir no opcionales como en el caso anterior, con la palabra reservada
// required
mostrarNombreYApellidos2({ required String nombre,
                        required String apellidos}) {
        print('$nombre $apellidos');
}
```

CONTROL DE FLUJO

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

- **if y else**
- **for (bucles)**
- **while y do-while (bucles)**
- break y continue
- **switch y case**
- assert

Ejemplos:

```
void main(){
        // CONDICIONALES
        // if-else
        print('-------------if-else-------------');
        if(2 > 3){
                print('2 es mayor que 3');
        }else if( 2 >4){
                print('2 es mayor que 4');
        }else {
                print('2 no es mayor que 3 ni 4');
        }

        // switch
        print('-------------switch--------------');
        String estado = 'abierto';
        switch(estado){
                case 'abierto':
                        print('Abierto');
                        break;
                case 'cerrado':
                        print('Cerrado');
                        break;
        }

        // BUCLES
        // for básico
        print('----------------for-----------------');
        var animales = ['perro', 'gato', 'tigre', 'elefante'];
        for( var i = 0; i < animales.length; i++){
                print(animales[i]);
        }
```

```
        // forEach
        print('-------------forEach--------------');
        animales.forEach((animal) {print(animal)});

        //otra forma equivalente para cuando solo es una línea
        animales.forEach((animal) => print(animal));


        // for-in
        print('-------------for-in--------------');
        for(var animal in animales){
                print(animal);
        }

        // while
        print('-------------While--------------');
        bool encontrado = false;
        int i = 0;
        while(!encontrado){
                if(animales[i] == 'elefante'){
                        print('Encontrado en $i');
                        encontrado = true;
                }
                i++;
        };

        // do while
        print('-------------do while--------------');
        encontrado = false;
        i = 0;
        do{
                if(animales[i] == 'elefante'){
                        print('Encontrado en $i');
                        encontrado = true;
                }
                i++;
        }while(!encontrado);
}
```

CLASES

Como ya hemos indicado anteriormente, Dart es un lenguaje orientado a objetos, por lo que en esta sección veremos cómo crear clases y sus métodos constructores, getters y setters.

```
void main(){
        // Para crear un instancia de nuestra clase
        final Animal animal = new Animal(); // new es opcional
```

```
        animal.nombre= 'Manolo';
        animal.especie = 'ave';

        print(animal);
        print(animal.nombre);
        print(animal.especie);
}

// Para declarar una clase se utiliza la palabra reservada class seguido
// del nombre de la clase (Mayúsculas y CamelCase)
class Animal {
        //PROPIEDADES, ojo al null safety!
        String? nombre;
        String? especie;

        // El contructor no tiene palabra reservada, es el propio nombre
        // de la clase.
        // Si implementamos el constructor ya no detecta nuestras
        // propiedades como null safety
        Animal(this.nombre, this.especie);

                //Esto es equivalente a:
                Animal(String nombre, String especie){
                        this.nombre = nombre;
                        this.especie = especie;
                }

        //También tenemos los métodos mágicos (getter, setter, toString,…)
        @override //El override es opcional
        String toString() {
                return 'nombre: ${this.nombre}';
                //se utilizan las {} para delimiter toda la variable
        }
}
```

Constructores con nombre, se utilizan cuando queremos crear una instancia, pero sin tener que enviarle las propiedades formateadas:

```
void main(){
        // Los constructores con nombre sirven cuando yo en vez de recibir
        // las propiedades, recibo un Map, esto va a ser muy común en Flutter
        // Tendremos Maps que vendrán de Json.
        final Map<String, String> rawJson = {
                'nombre': 'Manolo',
                'especie': 'Pez'
        };
        // Lo que buscamos es poder crear una instancia de la clase animal
        // enviándole directamente el Map
        Animal animal = new Animal.fromJson(rawJson);
}
```

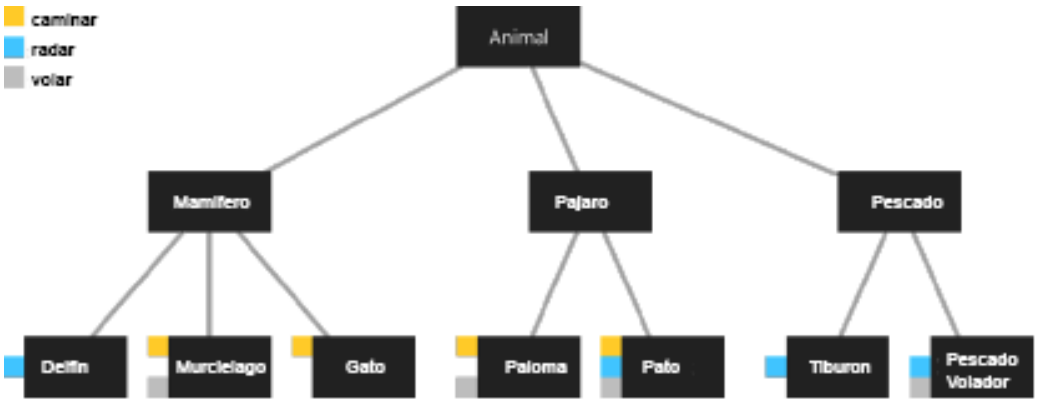
```
class Animal {
        //Las variables deben ser privadas, ponemos _ delante
        String _nombre;
        String _especie;

        //Constructor con nombre ( Animal.nombrequequeramos)
        // lo llamamos fromJson por convención
        Animal.fromJson( Map<String, String> json){
                this._nombre = json['nombre'] ?? '',
                this._especie = json['especie']!
        };
        //el ?? es un ternario, quiere decir que si no recibe nada lo
        //ponga vacío
        //la ! hace que no sea obligatorio el null safety

        //Generamos los getters y setters
        String get nombre => _nombre ?? '';
        set nombre(String value) {
                _nombre = value;
        }

        String get especie => _especie ?? '';
        set especie(String value) {
                _especie = value;
        }
}
```

MIXINS

Puesto que Dart no permite extender de más de una superclase, entran en juego los **MIXINS**.

En resumen, es la forma de extender de una superclase principal y a su vez tener acceso a métodos o propiedades de otras clases más específicas.

Vamos a verlo con un ejemplo:



Tienes una superclase llamada Animal, la cual tiene tres subclases (Mamiferos, Pajaros y Peces). En la parte inferior, tienes clases concretas. Los cuadros pequeños representan el comportamiento. Por ejemplo, el cuadro azul indica que una instancia de una clase con este comportamiento puede nadar.

Algunos animales comparten un comportamiento en común: Un gato y una paloma ambos pueden caminar; pero el gato no puede volar.

Este tipo de comportamiento es ortogonal a la clasificación, entonces no puedes implementar este comportamiento en las superclases.

Si una clase pudiera tener más de una superclase, podría ser fácil, crearías tres clases: Caminante, Nadador, Volador. Luego de esto, solo tendrías que heredar Paloma y Gato desde la clase Caminante. Pero en Dart, cada clase (excepto por **Object**) tiene exactamente una superclase.

En lugar de heredar desde la clase Caminante, podrías implementarla como si fuera una interfaz, pero deberías implementar el comportamiento en múltiples clases, entonces no es una buena solución.

Necesitas una forma de reutilizar código de una clase en múltiples jerarquías de clases. Los Mixins son exactamente eso. ***Los Mixins son una forma de reutilizar código de una clase en múltiples jerarquías de clase.***

SINTAXIS

Los mixins son definidos implícitamente a través de declaraciones de clase ordinarias:

```
class Caminante {
  void caminar() {
    print("Soy un caminante");
  }
}
```

Si quieres evitar que el MIXIN sea instanciado o extendido, puedes definirlo así:

```
abstract class Caminante {
        void caminar() => print('estoy caminando');
}
```

Para usar un MIXIN, use la palabra clave ***with*** seguido de uno o más nombres de MIXIN:

```
class Gato extends Mamifero with Caminante {}
class Paloma extends Pajaro with Caminante, Volador {}
```

Definiendo el MIXIN Caminante en la clase Gato, te permite llamar el método caminar, pero no el método volar (definido en Volador).

```
void main() {
  Gato gato = Gato();
  Paloma paloma = Paloma();

  //Un gato puede caminar.
  gato.caminar();

  // Una paloma puede caminar y volar.
  paloma.caminar();
  paloma.volar();

  // Un gato normal no puede volar
  // gato.volar(); // Uncommenting this does not compile.
}
```

¿CUÁNDO USAR MIXINS?

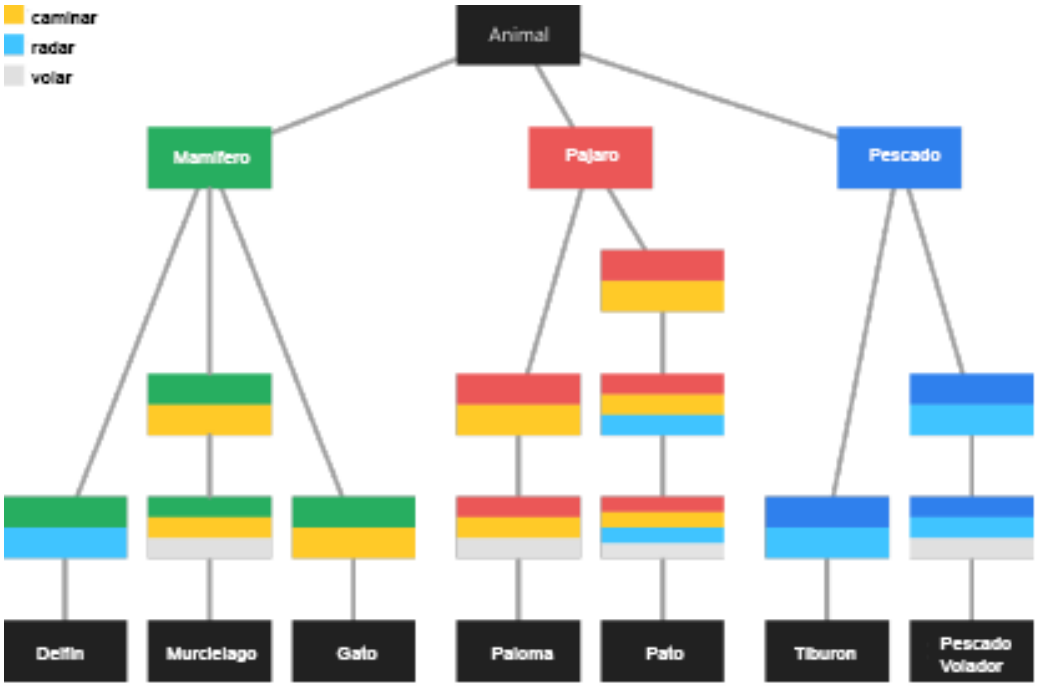
Los MIXINS son muy útiles cuando quieres compartir un comportamiento a través de múltiples clases que no comparten la misma jerarquía de clase, o cuando no tiene sentido implementar tal comportamiento en una superclase.

EJEMPLO ANIMAL COMPLETO

```
abstract class Animal { }
abstract class Mamifero extends Animal { }
abstract class Ave extends Animal { }
abstract class Pez extends Animal { }
abstract class Volador {
        void volar() => print('estoy volando');
}
abstract class Caminante {
        void caminar() => print('estoy caminando');
}
abstract class Nadador {
        void nadar() => print('estoy nadando');
}
```

```
class Delfin extends Mamifero with Nadador {}
class Murcielago extends Mamifero with Caminante, Volador {}
class Gato extends Mamifero with Caminante {}
class Paloma extends Ave with Caminante, Volador {}
class Pato extends Ave with Caminante, Volador, Nadador {}
class Tiburon extends Pez with Nadador {}
class PezVolador extends Pez with Nadador, Volador {}

void main() {
        // final flipper = new Delfin();
        // flipper.nadar();
        // final batman = new Murcielago();
        // batman.caminar();
        // batman.volar();
}
```

Puedes ver abajo como estos MIXINS son linealizados



SOPORTE ASÍNCRONO

FUTURES

Podríamos decir que los FUTURES son el equivalente en JavaScript a las promesas, o los "tasks" en C#. Es indispensable entenderlos y aprender a trabajar con ellos. En general un **FUTURE** no es más que una tarea asíncrona, es decir, algo que se va a ejecutar a diferente tiempo. Algo muy común de un FUTURE son las peticiones HTTP para acceder a la base de datos, lo vemos con un ejemplo:

```
void main() {
        //Primero se ejecutará los dos prints del main y a las 3 segundos el
        // print del final
        print('Antes de la petición');

        httpGet('https://inventada.com').then((data) => print(data));

        print('Fin del programa');
}

Future<String> httpGet( String url ) {
        //delayed necesita el tiempo de espera y que hará después
        return Future.delayed(
                Duration( seconds: 3 ), () =>'Hola Mundo - 3 segundos'
        );
}
```

ASYNC AWAIT

Es lo contrario del **FUTURE**. Hacemos una llamada asíncrona, se espera que esa llamada devuelva algo y luego se sigue ejecutando el código escrito a continuación.

```
void main() async {
        print('Antes de la petición');

         final data = await httpGet('https://inventada.com');
         print( data );

         print('Fin del programa');
}

//para poder hacer la llamada en un await el método debe ser asíncrono
//tambien
Future<String> httpGet( String url ) {
      return Future.delayed(
        Duration( seconds: 3 ), () =>'Hola Mundo - 3 segundos'
      );
}
```