Introducción a Dart

Dart es un lenguaje de programación tipado y orientado a objetos, desarrollado por Google. Fue presentado en el 2011 y su primera versión estable fue lanzada en el 2013. Dart es un lenguaje de programación de código abierto, con licencia BSD, y tiene como objetivo principal ser un lenguaje rápido, fácil de aprender, escalable y con capacidad de ejecutarse tanto en el lado del cliente como en el servidor.

Aunque se usa principalmente en el desarrollo de aplicaciones moviles con Flutter.

Variables

Las variables en Dart se declaran con la palabra reservada `var` seguido del nombre de la variable y el valor que se le asignará.

```
var nombre = 'Diego';
var edad = 23;
```

Tambien podemos definir una variable asignandole su tipo de dato.

```
String nombre = 'Diego';
int edad = 23;
```

Tambien podemos declarar variables con `final`, el valor de estas variables solo puede ser declarado una vez y luego no puede ser modificado.

```
final String nombre = 'Diego';
final int edad = 23;
```

Podemos delcara variables tambien usando `late`, este tipo de varibles se inicializan cuando se usan por primera vez.

```
late String nombre;
late int edad;
```

Tipos de datos

En Dart tenemos los siguientes tipos de datos:

- `int`: Números enteros.
- `double`: Números decimales.

- `String`: Cadenas de texto.
- `bool`: Valores booleanos.
- `List`: Listas.
- `Map`: Mapas.
- `dynamic`: Cualquier tipo de dato.

Por ejemplo:

```
int edad = 27;
double estatura = 1.75;
String nombre = 'Diego';
bool activo = true;
List<String> amigos = ['Miguel', 'Pedro', 'Luis'];
Map<String, dynamic> persona = {
  'nombre': 'Diego',
  'edad': 23,
  'estatura': 1.75,
  'activo': true
};
dynamic variable = 23;
```

Estructuras de colecciones Lists, Maps, Iterables, Sets

Las `Lists` son el equivalente al array en otros lenguajes, estas se caracterizan por tener llaves cuadradas `[]` y pueden contener cualquier tipo de dato.

```
List<String> amigos = ['Miguel', 'Pedro', 'Luis'];
```

Las listas tambien tienen sus metodos como por ejemplo:

- `length`: Nos devuelve la cantidad de elementos que tiene la lista.
- `first`: Nos devuelve el primer elemento de la lista.
- `reversed`: Nos devuelve la lista en reversa.

Este último metodo nos devuelve un iterable. Un `Iterable` es una colección de elementos que se puede leer de manera secuencial. Es un objeto que puede contar elementos que se encuentren dentro de él, como listas. sets, arreglos, etc. Se caracteriza por devolver una especie de lista pero con parentesis `()`.

```
List<String> amigos = ['Miguel', 'Pedro', 'Luis'];
print(amigos.length); // 3
print(amigos.first); // Miguel
print(amigos.reversed); // (Luis, Pedro, Miguel)
```

```dart
final numbers = [1, 2, 3, 4, 5];

final reversed = numbers.reversed;

// para recuperar la lista original
final reversedList = reversed.toList();
```

Los `Sets` son colecciones de elementos unicos, es decir, no pueden existir elementos repetidos dentro de un set. Se caracterizan por tener llaves `{}`.

```dart
Set<String> amigos = {'Miguel', 'Pedro', 'Luis'};
```

Funciones y parámetros

En Dart tenemos dos tipos de funciones, las funciones con nombre y las funciones anonimas.

Las funciones con nombre se caracterizan por tener un nombre y un cuerpo, el cuerpo de la función es el bloque de código que se ejecuta cuando llamamos a la función.

```dart
void saludar() {
  print('Hola');
}
```

Las funciones anonimas se caracterizan por no tener un nombre, solo un cuerpo. Estas funciones se pueden almacenar en una variable y luego ejecutarlas.

```dart
final saludar = () {
  print('Hola');
}
```

Tambien tenemos las funciones de flecha, estas son una forma mas corta de escribir una función anonima.

```dart
String saludar() => 'Hola';

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

Tenemos también el checkeo de parametros nulos y opcionales.

```dart
int addTwoNumbersOptional(int a, [ int? b ]) { // [] representa que el
parametro es opcional y ? representa que el parametro puede ser nulo
```

```
  b = b ?? 0; // si el parametro b es nulo, se le asigna el valor 0

  return a + b; // de lo contrario dart nos arrojará un error
}
```

Podemos tambien hacer los mismo de una manera mas corta:

```
int addTwoNumbersOptional(int a, [int b = 0]) {
  return a + b;
}
```

De esta forma establecemos que el valor por defecto de b será 0.

Clases y objetos

```
void main() {

  final Hero wolverine = new hero(name: 'Logan',power: 'Regeneración');

  print(wolverine.name);

}

class hero {
  String name;
  String power;

  Hero({required this.name, required this.power});
}
```

@override

Podemos sobre escribir el comportamiento de un metodo, para ello usamos la anotación `@override`.

```
class Animal {
  String nombre;

  Animal({required this.nombre});

  void emitirSonido() {
    print('Animal');
  }
}

class Perro extends Animal {
  String raza;
```

```
  Perro({required this.raza, required nombre}) : super(nombre: nombre);

  @override
  void emitirSonido() {
    print('Guauuuu');
  }
}
```

De esta forma, cuando llamemos al metodo `emitirSonido` de la clase `Perro`, se ejecutará el metodo `emitirSonido` de la clase `Animal`.

Constructores con nombre

```
class Hero {

  String name;
  String power;
  bool isAlive;


  Hero({required this.name, required this.power});

  Hero.fromJson( Map<String, dynamic> json ) : name = json['name'] ?? 'Sin
nombre', power = json['power'] ?? 'Sin poder', isAlive = json['isAlive'] ??
false;

  @override
  String toString() {
    return 'Heroe: $name - Poder: $power - isAlive: ${ isAlive ? 'Está vivo' :
'Está muerto' }';
  }

}
```

Getters y Setters

```
void main() {

  final mySquare = Square(side: 10.0);

  print(mySquare.area);

}

class Square {
  double _side; // El guion bajo es una convención para indicar que la variable
```

```
es privada

  Square({ required double side }): _side = side;


  double get area {
    return _side * _side;
  }

  set side(double value) {
    print('El valor del lado es: $value');
    if (value <= 0) {
      throw('El lado no puede ser menor o igual a 0');
    }
    _side = value;
  }
}
```

Aserciones

Las aserciones son una forma de validar que un valor cumpla con ciertas condiciones, si no se cumple, dart nos arrojará un error.

```
void main() {

  final mySquare = Square(side: 10.0);

  print(mySquare.area);

}

class Square {
  double _side; // El guion bajo es una convención para indicar que la variable
es privada

  Square({ required double side }): assert(side >= 0, 'side must be >= 0'),
_side = side; // validamos que el valor de side sea mayor o igual a 0, de lo
contrario dart nos arrojará un error

  double get area {
    return _side * _side;
  }

  set side(double value) {
    print('El valor del lado es: $value');
    if (value <= 0) {
      throw('El lado no puede ser menor o igual a 0');
    }
    _side = value;
```

```
  }
}
```

Clases abstractas y enumeraciones

Una clase abstracta es una especie de molde que nos sirve para crear otras clases, pero no se pueden crear instancias de esta clase.

```
void main() {
}

enum PlantType {
  nuclear,
  solar,
  wind,
  hydroelectric
}

abstract class EnergyPlant {

  double energyLeft;
  PlantType plantType;

  EnergyPlant({required this.energyLeft, required this.plantType});

  void consumeEnergy(double amount); {
    trow UnimplementedError();
  }

  class WindPlant extends EnergyPlant {

    WindPlant({ required double initialEnergy }) : super(energyLeft:
initialEnergy, plantType: PlantType.wind);

    @override
    void consumeEnergy(double amount) {
      energyLeft -= amount;
    }

  }

}
```

Los enumeradores son un tipo de dato que nos permite definir un conjunto de constantes con nombre.

La principal diferencia entre `extends` e `implements` es que `extends` nos permite heredar los metodos y propiedades de una clase, mientras que `implements` nos obliga a implementar los metodos de una clase.

```
void main() {

  final nuclearPlant = NuclearPlant( energyLeft: 1000.0 );

  print('nuclear:  ${ chargePhone( nuclearPlant ) }');

}

double chargePhone( EnergyPlant plant ) {

  if( plant.energyLeft < 10 ) {
    throw('No hay suficiente energía');
  }

  return plant.energyLeft - 10;
}

enum PlantType { nuclear, wind, water }

abstract class EnergyPlant {

  double energyLeft;
  PlantType plantType;

  EnergyPlant({required this.energyLeft, required this.plantType});

  void consumeEnergy(double amount); {
    throw UnimplementedError();
  }

}

class NuclearPlant implements EnergyPlant {

  @override
  double energyLeft;

  @override
  final PlantType type = PlantType.nuclear;

  NuclearPlant({ required this.energyLeft });

  @override
  void consumeEnergy(double amount) {
    energyLeft -= ( amount * 0.5 );
  }

}
```

Mixins

Los mixins son una forma de reutilizar codigo en dart, estos son clases que no pueden ser instanciadas, pero si pueden ser heredadas.

```
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 {}
```

En resumen los mixins permiten añadir cierta funcionalidad a las clases de manera especifica.

Futures

Un `Future` representa principalmente el resultado de una operación asíncrona. Es una promesa de que pronto tendrás un valor. La promesa puede fallar y hay que manejar la excepción. Los `futures` son un acuerdo de que en el futuro tendrás un valor para ser usado.

```
void main() {
  print('Inicio del programa');

  httpGet('https://asd.com').then((value) {
    print(value);
  }).catchError((error) {
    print(error);
  });
```

```
  print('Fin del programa');
}

Future<String> httpGet(String url) {
  return Future.delayed(const Duration(seconds: 1), () {
    throw 'Error en la peticion http';
    //return 'Respuesta de la peticion http';
  });
}
```

Async y Await

La palabra reservada `async` se usa para indicar que una función es asincrona, que devuelve un `Future`, por tanto va a tardar un poco en devolver un resultado.

La palabra reservada `await` se usa para indicar que una función debe esperar a que se resuelva un `Future` para continuar con la ejecución del programa.

```
void main() async {
  print('Inicio del programa');

  try {
    final value = await httpGet('https://asd.com');
    print(value);

  } catch (e) {
    print(e);
  }


}

Future<String> httpGet(String url) async {
  await Future.delayed(const Duration(seconds: 1));
  return 'Respuesta de la peticion http';
}
```

Try, on, catch y finally

```
void main() async {
  print('Inicio del programa');

  try {
    final value = await httpGet('https://asd.com');
    print(value);

  } on TimeoutException catch (e) {
```

```
    print('Timeout');
  } catch (e) {
    print(e);
  } finally {
    print('Fin del programa');
  }


}

Future<String> httpGet(String url) async {
  await Future.delayed(const Duration(seconds: 1));
  return 'Respuesta de la peticion http';
}
```

Streams

Los streams pueden ser retornados y usados como objetos, funciones o métodos, son un flujo de información que pueden estar emitiendo valores periódicamente, una única vez o nunca.

Un `Stream` podría verse como una mangera conectada a un tubo de agua, cuando abres el tubo el agua fluye, cada gota de agua sería una emisión del Stream, la mangera puede nunca cerrarse, cerrarse o nunca abrirse.

```
void main() {
  emitNumbers().listen((value) {
    print(value);
  });
}

Stream<int> emitNumbers() {
  Stream.periodic( const Duration(seconds: 1), (value) {
    return value;
  }).take(5);
}
```

async* y await

El `async*` lo que simboliza es que se va a retornar un stream.

```
void main() {
  emitNumbers().listen(value) {
    print('Stream value: $value');
  }
}

Stream emitNumbers() async* {
```

```
  final valuesToEmit = [1,2,3,4,5];

  for(int i in valuesToEmit) {
    await Future.delayed(const Duration(seconds: 1));
    yield i;
  }

}
```