La extensión de los archivos es ‘.dart’

Se le sulen por nombres a estos archivos en [Snake case](https://es.wikipedia.org/wiki/Snake_case)

Es muy similar a java

Sintaxis general

Todo termina en ; no es opcional

# main

Toda aplicación de dart ejecuta una función llamada main

void main(){}

Esta función puede síncrona o asíncrona

# for

Ejemplo de función con ciclo

void main(){

  for (var i = 0; i < 10; i++) {

    print('hola ${i}')

  }

}

## Avanzado

final valores = [1, 23, 4, 5, 6];

  for (var numero in valores) {

    print('el numero es $valores');

  }

# Tipos:

var: cualquier tipo

bool

similares a java

String // con mayúsculas

void,

## dynamic: cualquier tipo

por defecto acepta nulos

no infiere el tipo, simplemente siempre va a ser de tipo dynamic

## nuleable

Al igual que en C# se pueden declarar valores básicos que puedan admitir valores nulos agregándole al tipo un ’?’

bool? b;

bool? b = null;

En ese caso ambos equivalen a lo mismo, las variables no declaradas se asumen como nulas

## final

final d="CONSTANTE";

## late

late final d="CONSTATNE";

el subtipo late es una inicialización tardía

Puede que en algunos casos no se disponga del valor (y se conozca que mas adelante lo va a tener) de una variable pero es necesario que se trate en el código como que siempre lo tiene

## Const

const d="CONSTANTE";

## Sin tipado

Si es usa final o const se puede declarar la variables sin tiparlas porque asume el tipo cuando se instancian

const d="CONSTANTE";

final d="CONSTATNE";

## listas

parecido a Python es con los []

const e=['a','b']

### usarlas

print("${e.first}");

  print("${e.isEmpty}");

  print("${e.isNotEmpty}");

  print("${e.last}");

  print("${e.length}");

  print("${e.firstOrNull}");

final Iterable<String> ite4=lista.reversed;

 Iterable<int> ite6 = lista2.where((num) {

    return num > 2;

  });

indicándole el tipo al [] al usar genericos

const e2 = <String>['a', 'b'];

const List<String> e2 = <String>['a', 'b'];

const List<String> e2 = ['a', 'b'];

## set de datos

const f = {1, 23, 4};

const Set<int> f = <int>{1, 23, 4};

## mapas

const mapa = {'key1': 'valor1', 'key2': 'valor2'};

const Map<String,String> mapa2=<String,String>{'key1': 'valor1', 'key2': 'valor2'};

### usarlos

final valor2 = mapa['key1'];

no se puede usar en este tipo de declaración el tipo const

## iterables

const ite = ('a', 'b', 'c');

const (String,String,String) ite2 = ('a', 'b', 'c');

son como las tuplas de python

### iterables que son mas iterables

final ite3=lista.reversed;

final Iterable<String> ite4=lista.reversed;

tienen mas métodos que los que solo están declarados como ()

print("${ite3.toList()}");

  print("${ite3.toSet()}");

# Texto con formato (interpolación de strings):

Entre comillas se puede incluir un signo de dólar seguido de unas llaves para incluir un contenido dinámico

print('hola ${i}')

Tambien solo se puede ejecutar código como métodos si están entre comillas

print('toUpperCase ${ variable.toUpperCase()}');

O directamente una variable si se usa un signo de dólar, solo variables directamente, no sus metodos

dynamic a='asd $b';

# funciones de string

los objetos de tipo string tienen métodos

print('toUpperCase ${variable.toUpperCase()}');

  print('toLowerCase ${ variable.toLowerCase()}');

  print('length ${ variable.length}');

  print('isEmpty ${ variable.isEmpty}');

  print('isNotEmpty ${ variable.isNotEmpty}');

# String multilinea

Al igual que python se puede utilizar

  String a = """

linea 1

linea 2

""";

Se puede usar junto a la interpolación de strings

  String a = """

linea 1 $myName

linea 2

""";

# Declarar funciones

Pueden no tener un tipo declarado

metodoDePrueba() {

  print("hola");

}

## Sintaxis Funciones

Tipo\_retorno nombre\_funcion(){}

## Externas al main

Se pueden declara funciones en el mismo archivo del main

void main() {

  metodoDePrueba();

}

metodoDePrueba() {

  print("hola");

}

## Dynamic automatico

Si no se declara un tipo de retorno, automáticamente el tipo de retorno siempre va a hacer dynamic

metodoDePrueba2() {

  return 'hola';

}

## Call back

Se utilizan dentro de otros métodos, son el equivalentes los lamdas o funciones de flecha

print("${lista2.where((num) {

    return num > 2;

  })}");

El parámetro se puede tipar

Iterable<int> ite6 = lista2.where((int num) {

    return num > 2;

  });

### Funciones de flecha

Se pueden declarar si son cortas en un mismo archivo

void main() {

  print(metodoDePrueba());

}

String metodoDePrueba() => "hola";

int metodoDePrueba4(int a, int b) => a + b;

## Parámetros

Si no se declara el tipo se ponen automáticamente en Dynamic

int metodoDePrueba3(a, b) {

  return a + b;

}

### Se pueden tipar

int metodoDePrueba3(int a, int b) {

  return a + b;

}

### Pueden ser opcionales

Los opciones se declaran dentro de unos [] y deberían ser nulables si no tienen valor por defecto

metodoDePrueba5(int a, [int? b]) {

  print("$a $b");

}

metodoDePrueba6(int a, [int b = 2]) {

  print("$a $b");

}

### Opcionales por nombre

La idea es similar a los keysargs de python

Se ponen entre {}

metodoDePrueba7(int a, int b, {String? nombre, String? apellidos}) {

  print("$a $b");

}

metodoDePrueba7(1, 2, apellidos: 'algo');

#### Requeridos

En este caso el primero es requerido

metodoDePrueba8(int a, int b, {required String? nombre, String? apellidos}) {

  print("$a $b");

}

# Operadores de valores nulos

int? a;

int b = a ?? 1;

en caso de ser nula asigna el valor de la derecha

int? a; // = null

  a ??= 3;

  print(a); // <-- Imprime 3

  a ??= 5;

  print(a); // <-- Aún imprime 3

  print(1 ?? 3); // <-- Imprime 1.

  print(null ?? 12); // <-- Imprime 12.

# Operador suma y resta

Es valido

b++;

b--;

# Clases

## Declararlas

class Hero {

  String? name;

  String? power;

}

### atributos

Los atributos si no se declaran tienen que ser nuleables

#### Públicos

Son los que se declaran de forma normal y pueden ser accedidos desde afuera

class Hero {

  String name;

  String power;

  Hero({required this.name, this.power = "sin poder"});

}

final a = new Hero(name: "a", power: "b");

  a.name = "otro nombre";

#### Privadas

Los atributos privados tienen un ‘\_’ delante y no se pueden usar directamente en la declaración por nombre del constructor, tienen que ser inicializados después

class Scuare {

  double \_side;

  Scuare({required side}) : \_side = side;

  double get area {

    return \_side \* \_side;

  }

}

#### propiedades

parecido a C#

##### get

void main() {

  final square = Scuare(side: 10);

  print("el area es: ${square.area}");

}

class Scuare {

  double side;

  Scuare({required this.side});

  double get area {

    return side \* side;

  }

}

##### Set

class Scuare {

  double \_side;

  Scuare({required side}) : \_side = side;

  double get area {

    return \_side \* \_side;

  }

  set side(double value) {

    print('setting new value $value');

    if (value < 0) throw 'el valor no puede ser menor a 0';

    \_side = value;

  }

}

void main() {

  final square = Scuare(side: 10);

  print("el area es: ${square.area}");

  square.side = 5;

}

### Constructor

#### Con :

Los atributos que no sean nuleables tienen que instanciarse en el principio del constructor luego de los ‘:’,

creo que en ese caso los parámetros deben de tener distinto nombre del de los atributos

class Hero {

  String name;

  String power;

  Hero(String name\_atributo, String power\_atributo)

      : name = name\_atributo,

        power = power\_atributo;

}

#### Con this sin cuerpo

class Hero {

  String name;

  String power;

  Hero(this.name, this.power);

}

#### Por nombre

class Hero {

  String name;

  String power;

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

}

void main() {

  final a = new Hero(name: "a", power: "b");

  final b = Hero(name: "a", power: "b");

}

##### Con valores default

class Hero {

  String name;

  String power;

  Hero({required this.name, this.power="sin poder"});

}

No necesita el required

#### fromJson

class Hero {

  String name;

  String power;

  Hero({required this.name, this.power = "sin poder"});

  Hero.fromJson(Map<String, dynamic> json)

      : name = json["name"] ?? "sin nombre",

        power = json['power'];

}

Se usa

void main() {

 final Map<String, dynamic> variableRawJson = {

    'name': 'el nombre',

    'power': 'el poder'

  };

  final heroe = Hero.fromJson(variableRawJson);

}

### toString

class Hero {

  String name;

  String power;

  Hero({required this.name, this.power = "sin poder"});

  String toString() {

    return "valor string";

  }

}

#### lamda

class Hero {

  String name;

  String power;

  Hero({required this.name, this.power = "sin poder"});

  String toString() => "valor string";

}

#### Llamando atributos sin el this

class Hero {

  String name;

  String power;

  Hero({required this.name, this.power = "sin poder"});

  @override

  String toString() {

    return "$name - $power";

  }

}

### @override

Es una buena practica ponerlo para que se sepa que se esta sobrescribiendo un método, (pero no es obligatorio)

class Hero {

  String name;

  String power;

  Hero({required this.name, this.power = "sin poder"});

  @override

  String toString() {

    return "valor string";

  }

}

### Funciones internas

Son similares a las externas

class Hero {

  String name;

  String power;

  Hero({required this.name, this.power = "sin poder"});

  Hero.fromJson(Map<String, dynamic> json)

      : name = json["name"] ?? "sin nombre",

        power = json['power'];

  String nombreCompleto() {

    return "$name - $power";

  }

  @override

  String toString() {

    return "$name - $power";

  }

}

## Instanciarlas

final a = new Hero();

  final b = Hero();

no es necesario usar el new

## Abstractas

Obviamente no se pueden instanciar

abstract class EnergyPlant {

  PlantType type;

  EnergyPlant({required this.type});

  void cosumeEnergy(double amount);

}

Se le declaran métodos abstractos

### Implementarlas

Las clases hijas tienen que implementar los métodos abstractos

class WindPlant extends EnergyPlant {

  WindPlant({required double initialEnergy})

      : super(energyLeft: initialEnergy, type: PlantType.wind);

  @override

  void cosumeEnergy(double amount) {

    energyLeft -= amount;

  }

}

#### implements

La implementación nos ayuda a nosotros a explícitamente colocar cada uno de sus   @override

class NuclearPlant implements EnergyPlant {

  @override

  double energyLeft;

  @override

  final PlantType type = PlantType.nuclear;

  NuclearPlant({required this.energyLeft});

  @override

  void cosumeEnergy(double amount) {

    energyLeft -= amount \* 0.5;

  }

}

## Herencia

class WindPlant extends EnergyPlant {

}

### Super

#### constructor

Se puede llamar al constructor de la super clase por medio de ‘super(argumentos para la superclases)’

class WindPlant extends EnergyPlant {

  WindPlant({required double initialEnergy})

      : super(energyLeft: initialEnergy, type: PlantType.wind);

}

## Mixin

Son (clases abstractas, antes de dart 3, ahora en ves de class es ‘mixin’) que pueden ser usadas en otra clase, a demás de su herencia (como las interfaces o multiples superclases )

Se usan en otra clase después de la palabra reservada ‘with’

No deben de tener un constructor

mixin Nadador {

  void nadar() {}

}

mixin Caminante {

  void caminar() {}

}

mixin Volador {

  void volar() {}

}

abstract class Animal {

  void comer() {}

}

abstract class Mamifero extends Animal {

  void amamantar() {}

}

class Delfin extends Mamifero with Nadador {}

class Murcielago extends Mamifero with Volador, Caminante {}

# Ternario

const c = true ? "a" : "b";

similar a java y javascripts

# Excepciones

## Dispararlas

### Errores

Como tal no llega a ser un objeto Excepcion

  if (1 > 2) throw 'mensaje del error';

### excepciones

if (1 < 2) throw Exception("el mensaje del error");

## assert

se declaran reglas en la inclinación de los métodos y el constructor para que arrojen errores si no se cumplen

### constructor

class Scuare {

  double \_side;

  Scuare({required side})

      : assert(side > 0),

        \_side = side;

}

### Con mensaje de error

Como segundo argumento al assert se le pone el mensaje

class Scuare {

  double \_side;

  Scuare({required side})

      : assert(side > 0, 'el valor debe ser superior a 0'),

        \_side = side;

}

## Try catch finally

try {

    await metodoAsincrono("valor de prueba");

  } catch (err) {

    print('error $err');

  } finally {

    print("se ejecuta siempre");

  }

### Por tipo

try {

    await metodoAsincrono("valor de prueba");

  } on HttpException {

    print("en este caso expesifico");

  } catch (err) {

    print('si no entra en otro error $err');

  } finally {

    print("se ejecuta siempre");

  }

Capturando la excepción

  try {

    await metodoAsincrono("valor de prueba");

  } on HttpException catch (err) {

    print("en este caso expesifico $err");

  } catch (err) {

    print(' si no entra en otro error $err');

  } finally {

    print("se ejecuta siempre");

  }

# Enum

Como las clases, no se le pone ; al final de la declaración

Y los tipos internos no son variables, son directamente valores escritos en ese sitio

enum PlantType { nuclear, wind, water }

como en todos los lenguajes aquí tambien representan un tipo

abstract class EnergyPlant {

  PlantType type;

  EnergyPlant({required this.type});

}

Se usa luego, como en java

PlantType tipo = PlantType.water;

# Future

Objetos para ejecutar promesas, donde se tiene que definir la función a ejecutar y el tipo de valor de retorno de esta asi como los parámetros que necesita, luego la función a ejecutar si tienen éxito con lo que procesa el resultado de la anterior y la función en caso de error

void main() {

  httpGet('direccion').then((value) {

    print(value);

  }).catchError((err) {

    print('error $err');

  });

}

Future<String> httpGet(String url) {

  return Future.delayed(Duration(seconds: 1), () {

    return 'Hello World';

  });

}

## Async

Los métodos asíncronos siempre tienen que retornar un Future, a pesar de que no se declare directamente en el return

Future<String> metodoAsincrono(String argumento) async {

  return 'Hello World';

}

## Await

Al declarar la llamada a una promesa con await, el código espera el resultado y toma directamente el valor de la función (no hay que usarlo con la llamada al then)

Future<String> metodoAsincrono2(String argumento) async {

  String resultado = await metodoAsincrono("otro valor");

  return 'Hello World 2';

}

Solo se puede usar dentro de funciones asíncronas y para llamadas a promesas

Se puede convertir al main en async

void main() async {}

## delayed

detiene el código un tiempo

Stream<int> emitNumbers2() async\* {

  for (var numero in [1, 2, 3, 4, 5]) {

    await Future.delayed(const Duration(seconds: 1));

    yield numero;

  }

}

# Stream

Son objetos que devuelven un flujo constante de datos

Stream<int> emitNumbers() {

  return Stream.periodic(const Duration(seconds: 1), (value) {

    return value;

  });

}

## Sin escucha

Si no se pone algo que escuche valores la llamada a la función seria como que ignorada? Osea el código no va esperar por ella y el programa va a terminar

void main() {

  emitNumbers();

}

## listen

Para escuchar los valores es con el metodo listen

void main() {

  Stream<int> flujo = emitNumbers();

  flujo.listen((value) {

    print("valor actual $value");

  });

}

El programa no va a terminar hasta que se intervenga externamente

## Take

Se puede limitar la cantidad de iteraciones que se produce usando para ello el metodo ‘take’ este retorna un Stream nuevo, al terminar esta cantidad el programa se detiene

Stream<int> flujo = emitNumbers();

  flujo = flujo.take(5);

  flujo.listen((value) {

    print("valor actual $value");

  });

## Yield

Sustituye al return (y es como en python) y retorna un valor parcial en el momento que se llama a la funcion, a la siguiente llamada la funcion continua por donde se detuvo hasta que llega al próximo yield

Debe de usarse en funciones async\* porque estas retorna datos de forma dinámica

Stream<int> emitNumbers2() async\* {

  for (var numero in [1, 2, 3, 4, 5]) {

    await Future.delayed(const Duration(seconds: 1));

    yield numero;

  }

}