Flutter

MINI APUNTES DART 1

Desarrollo de Interfaces

3 DE NOVIEMBRE DE 2024

ERIK AMO TOQUERO

Índice

Tipos de Variables en Dart

Dart permite declarar varios tipos de variables:

- **var**: Permite que las variables puedan tener diferentes tipos al momento de asignarse, es decir, da igual si se inicializa siendo int, double o TuMadre().
- **final:** Se asigna un valor que no puedes cambiar una vez que inicias una variable
- **Const:** Define una constante que no cambia durante la compilación

En Dart también se introduce dos tipos de valores nuevos: Future y Nullable

Valor Futuro

Este tipo de variable se utiliza en la asincronía para declarar una variable que **no se inicia instantáneamente, pero se iniciará por un proceso asíncrono a su flujo**. Para acceder a ellos luego se usa **await**.

Valor Nulo por Defecto (Nullable)

Para poder hacer valores nulos, se utiliza el operador nullable (?) a la hora de inicializar la variable. Esto permite no inicializar instantáneamente la variable al inicio del programa u objeto, sino iniciarlo más tarde en el programa.

```
String? prueba;
prueba = "Hola";
```

Si hay un método que devuelve un valor nullable, debemos asegurar que devuelve el valor correctamente con '!' al final:

```
String prueba = stdin.readLineSync()!;
```

Tipos de Datos

Parecido a Java:

- **int**: Entero
- **double**: Decimal
- **String**: Cadena de Caracteres
- **bool:** Booleana

También hay colecciones como en Java:

- **List<tipo>:** Listas ordenadas
- **Set<tipo>:** Listas no ordenadas y sin duplicados
- **Map<clave, valor>:** Listas de pares clave-valor
- **Dynamic:** Permite cambiar el tipo de dato.

```
dynamic variable = "Hola mundo"; //Es String
variable = 3; //Ahora int
variable = true; //Ahora booleana
```

Operadores:

Igual a Java

- **Aritméticos:** Suma +, resta -, multiplicación *, división /, resto de división %
- **Relacionales:** Menor (<) o igual (<=), mayor (>) o igual (>=), igual (==).
- **Lógicos:** And (&&), Or (||) y Not(!)

Estructuras de control sin más:

If-else y árboles de if-else-if

Funciona igual que en java

```
if(condicion){
   [...];
} else{
   [...];
}
```

Switch-case

Funciona igual que en java

```
var res;
switch (opt){
  case 1:
    res = num1+num2;
  case 2:
    res = num1-num2;
  case 3:
    res = num1*num2;
  case 4:
    res = num1/num2;
}
```

Otra alternativa es el switch-case declarando un valor, que no cambia mucho de java

```
var res = switch (opt) {
            1 => num1 + num2,
            2 => num1 - num2,
            3 => num1 * num2,
            4 => num1 / num2,
            _ => 0,};
```

'_' Es el valor por defecto (Default en Java), y la separación es por comas y no se pone Case en la segunda. En la segunda forma, en vez de poner un = se hace una flecha (=>)

While y Do-While

Igual que en Java

```
do{
   [...];
}while(condicion);
while(condicion){
   [...];
}
```

Control de excepciones

Para variar, igual que en Java

```
try{
}catch(excepcion){
} finally{
```

```
}
```

Entrada y salida de texto por línea de comandos

Salida

Tiene dos métodos: Print y por el import de dart:io

Print(String)

Funciona igual que el sout de Java:

```
print("String va aquí");
```

Stdout (dart:io)

La librería dart:io ofrece un escritor de línea de comandos por medio de su clase **stdout:**

```
stdout.write("String va aquí");
```

Entrada

Solo tiene un método de entrada, que es utilizando la librería dart:io. Esta ofrece su clase **stdin** para introducir texto:

```
stdin.readLineSync()
```

Este método no devuelve una variable normal, sino una variable **Nullable**

POO en Dart

La declaración de clases es igual que en todos los lenguajes:

```
class NombreClase{[...]}
```

A su vez, la declaración de objetos se parece más a Python y a Kotlin que a Java, pero por el mero hecho de que no usa el operador new

```
NombreClase nombreClase = NombreClase([parámetros]);
```

La llamada de métodos y atributos es igual que en Java (y en prácticamente todos los lenguajes)

```
nombreClase.metodo(parámetros);
nombreClase.atributo;
```

Constructores

Hay dos tipos de constructores: El constructor por defecto y los nombrados:

- Por defecto: Inicializan variables por parámetro (Constructor normal de java)
- Constructor nombrado: Ofrece distintas formas de inicializar el objeto:

```
NombreClase.soloNum1(int num){
  this.num = num;
  num2 = 0;
}
```

Herencia

Dart permite crear una clase base y extenderla. No hace falta poner Abstract u Open como en Java/Kotlin

```
class ClasePadre{[...]}
class ClaseHija extends ClasePadre{[...]}
```

Sobrecarga y MixIn

El polimiorfismo en Dart es mediante la sobreescritura/sobrecarga de métodos (Override) entre herencias entre clases

```
class ClasePadre{
  void hola(){
    print("Hola buenas");
  }
}
class ClaseHija extends ClasePadre{
  @override
  void hola(){
    print("Adios");
  }
}
```

Los Mixins permiten reutilizar código sin hacer herencia (el equiparable a Open de Kotlin)

```
mixin class ClasePadre{
  void hola(){
    print("Hola buenas");
  }
}
class ClaseHija with ClasePadre{
}

void main(){
  ClaseHija ch = ClaseHija();
  ch.hola();
}
```

Interfaces

Las interfaces no se parecen en nada a Java, ya que estas se puede implementar código en interfaces como una clase normal y corriente. Lo único que se utiliza el implements.

```
class ClaseInterfaz{
  void hola(){
    print("Hola Mundo");
  }
}
class ClaseMain implements ClaseInterfaz {
  @override
  void hola() {
    // TODO: implement hola
  }
}
```

Abstracción:

Las clases abstractes funcionan como en Java: No se pueden instanciar directamente y necesitan clases hijo que las utilicen

```
abstract class ClasePadre{
  void hola(){
    print("Hola Mundo");
  }
}
class ClaseMain extends ClasePadre{
  @override
  void hola() {
    // TODO: implement hola
  }
}

void main(List<String> args) {
  ClasePadre c = ClaseMain();
}
```

Enumeración

Permiten definir valores constantes que se pueden usar en el programa. Sirven para representar estados y tipos específicos

```
enum Direccion{
  norte, sur, este, oeste;
}
void main(List<String> args) {
  Direccion dir = Direccion.este;
  print("Vas hacia el ${dir.name}");
}
```

Colecciones

Las colecciones tienen distintas declaraciones y usos según el tipo de colección que sean:

List

Se declaran entre corchetes

```
List<String> usuarios = ["Pedro", "Paco", "María", "Juan"];
```

Set

Se declaran entre llaves

```
Set<String> usuarios = {"Pedro", "Paco", "María", "Juan"};
```

Map

Se declaran entre llaves, separando clave y valor por ‘:’

```
Map<int, String> usuarios = {1:"Pedro",2:"Paco",3:"María",4:"Juan"};
```

Todo lo demás de las colecciones funciona exactamente igual que en Java

Programación asíncrona

La programación asíncrona permite que haya funciones y atributos que funcionen de forma independiente a los tiempos del programa, tomando el tiempo necesario sin dar problemas de rendimiento. Dart utiliza Future, async y await para todo ello.

- **Future**: Tipo de variable que recibe una variable o función. Marca que al principio es nulo, pero en algún momento del futuro estará disponible
- **async:** Declaración de asincronía de una función. Define que una función es asíncrona, es decir, que en algún momento va a tener una función que requiera de una espera.
- **await:** Se utiliza para que el programa espere a que una función Future complete su ejecución

```
var url= Uri.https("img.freepik.com", "fotos-premium/hombre-dando-pulgar-arriba-pulgar-arriba_662214-43774.jpg");
Future<http.Response> futureresponse = http.get(url);
var response = await futureresponse;
if (response.statusCode==200){
  File file = File("Prueba.png");
  file.writeAsBytesSync(response.bodyBytes);
}
```

Esperar un tiempo – delayed

Para hacer un Thread.sleep() de Java, en Dart se hace con Future.delayed()

```
await Future.delayed(Duration(seconds: 3));
```

Streams

En español “Flujos”, son secuencias de eventos asíncronos en Dart. Sirven para manejar situaciones con muchos datos de por medio, por ejemplo, con la transmisión de datos de un servidor o eventos de usuario. Hay Streams de un solo suscriptor o de muchas escuchas (Broadcast).

```
Stream<int> contador = Stream.periodic(Duration(seconds: 1), (x) => x);
```

(En este caso Stream.periodic(periodo, funcion) sirve para ejecutar una orden cada x tiempo

El método que utiliza el stream para empezar a funcionar es listen(). Con él puede definir una función y definir acciones por si acaba, ya sea de forma correcta (onDone) o de forma fallida (onError)

```
Stream<int> contador = Stream.periodic(Duration(seconds: 1), (x) => x);
contador.listen((x){
  print(x);
},
onDone: () => print("Finalizado"),
onError: (e) => print("Error: ${e.toString()}"));
```

Librerías

Las librerías que ofrece Dart de primeras son:

- **dart:core:** Incorpora todas las funciones esenciales de Dart para crear aplicaciones
- **dart:math:** Incluye todo tipo de cálculos matemáticos
- **dart:async, dart:io, dart:convert…**

```
import 'dart:async';
```

Aparte, cuando Dart no tiene la librería por defecto, se pueden instalar librerías externas por medio de **paquetes**, los cuales se meten en pubspec.yaml

```
import 'package:http/http.dart' as http; //As da un nombre al paquete para usarlo correctamente
```

La mejor web para encontrar estos es: pub.dev/packages

La siguiente parte es el apartado de dependencias de pubspec

```
dependencies:
  args: ^2.4.2
  http: ^1.2.2
  image: ^4.3.0
  logger: ^2.4.0
  pdf: ^3.11.1
  csv: ^6.0.0
  encrypt: ^5.0.3
  htmltopdfwidgets: ^1.0.4
```

Esquema de un proyecto:

Se divide en archivos principales, configuración y carpetas comunes:

- **Archivos principales:**
  - **main.dart** -> Contiene el main del programa
- **Carpetas comunes:**
  - **lib/:** Contiene las clases y funciones del código fuente
  - **test/:** Tiene archivos de pruebas para asegurar la calidad del código
  - **bin/:** Archivos de scripts ejecutables
  - **web/:** Archivos para ejecutar el proyecto en la web
- **Configuración:**
  - **pubspec.yaml:** Gestiona las dependencias
  - **.gitignore**: Decide qué subir y qué ignorar en la versión de controles

Pubspec.yaml – Configuración

pubspec.yaml sirve para definir ciertos valores del proyecto:

- **name:** nombre
- **description:** descripción
- **version:** versión de proyecto
- **environment:** versión de sdk
- **dependencies:** lista de dependencias para el proyecto
- **dev_dependencies:** lista de dependencias solo para el desarrollo
- **flutter:** Opciones especificas para Flutter