Cábula Kotlin-Dart

As diferenças entre o Kotlin e o Dart são principalmente de teor sintático, sendo a conversão de uma para outra relativamente linear.

Abaixo apresenta-se uma tabela que permite, de forma sumarizada, perceber quais as primitivas da linguagem Dart que suportam os mecanismos que foram leccionados para a linguagem Kotlin.

Estrutura do programa

Kotlin	Dart	Observações

(ficheiro .kts)
Estrutura livre
(ficheiro .kt)
`fun main(){`
`  ...`
`}`
	(ficheiro .dart)
	`void main() {`
	`  ...`
	`}`
		Em Dart é obrigatório declarar uma função main. Em Kotlin, apenas é obrigatório declarar a função main se o ficheiro fôr .kt.
		Em Dart, os nomes dos ficheiros .dart não têm restrições, no entanto, costuma usar-se snake_case em vez de CamelCase como no Kotlin

`numero = 3`	`numero = 3;`	Em Dart, as linhas têm que terminar com ; (ponto e vírgula)

Declaração de variáveis

Kotlin	Dart	Observações

`var numero : Int = 10`	`int numero = 10;  (tipo explícito)`
	ou
	`var numero = 10; (tipo implícito)`
		Em Dart podemos declarar variáveis com tipo implícito ou explícito
		Nem todos os tipos estão disponíveis:

não existe; o kotlin é fortemente tipificado	`dynamic coisa = 10;  // neste momento é inteiro`
	`...`
	`coisa = 'ola'; // agora passou a ser string`
		Em Dart é possível mudar o tipo das variáveis.
		Deve ser usado com cuidado.

`var numero : Int (erro!!)`	`int numero; (erro !!)`	Tal como no Kotlin é obrigatório inicializar todas as variáveis em Dart.

`var numero : Int = 10 // mutável`
`val numero2 : Int = 11  // imutável`
`val numero2 : Int = getXYZ()  // imutável`
	`int numero = 10; // mutável`
	`const int numero2 = 11; // imutável`
	`final int numero3 = getXYZ(); // imutável`
		Em Dart, usa-se o const e o final para indicar que uma variável é imutável.
		A diferença é que no const, o valor tem que ser conhecido em tempo de compilação, enquanto no final pode ser conhecido apenas em tempo de execução

Kotlin	Dart
Byte	Não existe
Short	Não existe
Int	int
Long	não existe
Float	não existe
Double	double
Char	não existe
String	String
Boolean	bool

Nullability

Kotlin	Dart	Observações

`var numero : Int? = null`
`var numero2 : Double? = null`
`var texto : String? = null`
	`int? numero = null;`
	`double? numero2 = null;`
	`String? texto = null;`
		Tal como em Kotlin, todos os tipos podem ser nullable acrescentando um ? à frente do tipo.
		As variáveis nullable são inicializadas por omissão com o valor null

`var resultado = numero!! + 3`	`var resultado = numero! + 3;`	Para se usarem variáveis que podem ser null, e se tivermos a certeza que não têm o valor null, temos que pode-se colocar o !! (Kotlin) ou ! (Dart)
`var resultado = numero?.round()`	`var resultado = numero?.round();`	O safe operador (?) é igual em Kotlin e Dart
`var resultado = numero ?: 0`	`var resultado = numero ?? 0`	O elvis operator em Dart é representado por ??

Conversão de tipos

Kotlin	Dart	Observações
`var numero : Int = 3.4.toInt()` `var caracter : Char = 74.toChar()`	`var numero = 3.4.toInt();` `// não existe char`	Igual nas duas linguagens
`var numero : Int = "34".toInt()` `var numero2 : Double = "34.5".toDouble()`	`int numero = int.parse("34");` `double numero2 = double.parse("34.5");`	Em Dart, não existem primitivas toInt() e toDouble(). Para converter Strings para outros tipos existem primitivas int.parse() e double.parse()

Operações com Strings

Kotlin	Dart	Observações

`var texto = "Olá mundo"`
`var c = texto[4] // letra 'm'`
`var tamanho = texto.length`
	`var texto = "Olá mundo";`
	ou
	`var texto = 'Olá mundo';`
	`var c = texto[4] // letra 'm'`
	`var tamanho = texto.length();`
		Neste aspeto, o Kotlin e o Dart são praticamente iguais, exceptuando poder-se representar Strings com plicas em Dart

Arrays, Listas e Maps

Kotlin	Dart	Observações
`var numeros : Array<Int> = arrayOf(3, 6, 7)`	`não existe`	Em Dart, não existem arrays no sentido "clássico" - estruturas que têm que pré-alocadas com um certo tamanho e que não podem crescer posteriormente. em vez disso, usam-se listas.
`var numeros : List<Int> = listOf(3, 6, 7)`	`List<int> numeros = [3, 6, 7];`	A sintaxe para inicializar listas é mais compacta em Dart, basta colocar os elementos entre parêntesis retos.

`var numeros = listOf(3, 6, 7)`
`println(numeros[1])  // escreve 6`
`println(numeros.size) // escreve 3`
	`var numeros = [3, 6, 7];`
	`print(numeros[1]);  // escreve 6`
	`print(numeros.length) // escreve 3`
		A sintaxe para aceder aos elementos de uma lista é idêntica.

`var idadesPorNome: Map<String,Int> =` `mapOf("Pedro" to 40, "Sara" to 16)`	`Map<String,int> idadesPorNome =` `{"Pedro": 40, "Sara": 16};`	Os mapas/dicionários são representados pelo tipo Map em ambas as linguagens. No entanto, a forma de inicializar é mais compacta em Dart, com uma sintaxe similar ao Javascript (JSON)
`println(idadesPorNome["Pedro"])`	`print(idadesPorNome["Pedro"]);`	A forma de obter um valor a partir de uma chave é igual nas 2 linguagens

Escrita no écran

Kotlin	Dart	Observações

`println("olá")`	`print("olá");`
	
	ou
	
	`print('olá');`
		Em Dart, as Strings podem estar dentro de plicas ou de aspas.
		
		O print do Dart termina com uma quebra de linha. Se se quiser realmente fazer só um print sem quebra de linha tem que se usar a função stdout.write(...)

`println("numero = $numero")` `println("numero = ${numero}")`	`print("numero = $numero")` `print("numero = ${numero}")`	Para incluir variáveis dentro das Strings, o sistema é o mesmo.

Estruturas de controlo - selecção

Kotlin	Dart	Observações

`if (numero >= 10) {`
`  soma += numero`
`} else {`
`  soma = 10`
`}`
	`if (numero >= 10) {`
	`  soma += numero;`
	`} else {`
	`  soma = 10;`
	`}`
		Sintaxe igual, tirando os ponto e virgula

`if (numero in 1..10) {`
`  println("numero entre 1 e 10")`
`}`
	`if (numero >= 1 && numero <= 10) {`
	`  print("numero entre 1 e 10");`
	`}`
		Em Dart, não existe o operador `in`

`var positivo : Boolean = if (numero < 0) false else true`	`boolean positivo = numero < 0 ? false : true;`	Em Dart, existe o chamado operador ternário, que usa o ? e o : para construir uma espécie de if numa única linha: `<variável> = <condição> ? <valor-caso-condicao-true> : <valor-caso-condicao-false>`

`var texto1 : String = ...`
`if (texto1 == "ola") {`
`  ...`
`}`
	`String texto1 = ...;`
	`if (texto1 == "ola") {`
	`...`
	`}`
		Em Dart também se pode usar o operador == para comparar Strings

`var cor : String = ...`
 	
`when (cor) {`
`  "amarelo" -> println("amarelo")`
`  "azul" -> println("azul")`
`  else -> println("cor desconhecida")`
`}`
	`String cor = ...;`
	
	`switch (cor) {  // forma clássica`
	`  case "amarelo": print("amarelo");`
	`  case "azul": print("azul");`
	`  default: print("cor desconhecida");`
	`}`
		Em Dart, o equivalente ao when é o `switch`.
		
		Note-se que, ao contrário do C e do Java, não é necessário colocar um break no final de cada case.
		
		Em vez do default, pode-se usar o _

`var monocromatico = when (cor) {`
`  "amarelo", "azul", "vermelho" -> false`
`  "preto", "branco" -> true`
`  else -> null`
`}`
	`bool? monocromatico = switch(cor) {`
	`  "amarelo" || "azul" || "vermelho" => false,`
	`  "branco" || "preto" => true,`
	`  _ => null`
	`};`
		Em Dart também é possível usar o switch como expressão

Estruturas de controlo - repetição

Kotlin	Dart	Observações

`var i = 0`
`while (i < 10) {`
`  println(i)`
`  i++`
`}`
	`int i = 0;`
	`while (i < 10) {`
	`  print(i);`
	`  i++;`
	`}`
		Sintaxe igual, tirando os ponto e virgula

`var i = 0`
`do {`
`  println(i)`
`  i++`
`} while (i < 10)`
	`int i = 0;`
	`do {`
	`  print(i);`
	`  i++;`
	`} while (i < 10);`
		Sintaxe igual, tirando os ponto e virgula

Kotlin	Dart	Observações

`for (i in 1..10) {`
`  println(i)`
`}`
	`for (var i = 1; i <= 10; i++) {`
	`  print(i);`
	`}`
		Em Dart não existe o operador in nem os intervalos. Por isso, tem que se usar a forma clássica do `for` (como se usa na linguagem C), em que são definidas 3 partes separadas por ponto e vírgula:
		`for (inicialização; condição; incremento)`
		Este tipo de ciclo não existe em Kotlin

`var numeros = arrayOf(2, 6, 7)`
`for (numero in numeros) {`
`  println(numero)`
`}`
	`var numeros = [ 2, 6, 7 ];`
	`for (var numero in numeros) {`
	`  print(numero);`
	`}`
	ou
	`for (var i = 0; i < numeros.length; i++) {`
	`  print(numeros[i]);`
	`}`
		Sintaxe muito semelhante. Note-se que em Dart não existem arrays.
		Ou então pode-se usar a forma clássica do for (explicado no ponto anterior)

`var texto = "olá"`
`for (letra in texto) {`
`  println(letra)`
`}`
	`String texto = "olá";`
	`for (int i = 0; i < texto.length; i++) {`
	`  print(texto[i]);`
	`}`
		Em Dart não é possível percorrer a String diretamente, temos que usar o for clássico para ir obtendo os caracteres um a um

Funções

Kotlin	Dart	Observações

`fun soma(num1: Int, num2: Int) : Int {`
`  return num1 + num2`
`}`
	`int soma(int num1, int num2) {`
	`  return num1 + num2;`
	`}`
		A diferença de sintaxe reflecte a diferença como se declaram variáveis. De resto é semelhante.

`fun escreve(palavra: String) {`
`  println(palavra)`
`}`
	`void escreve(String palavra) {`
	`  print(palavra);`
	`}`
		Em Dart, as funções que não retornam nada devem usar o tipo `void`

`fun soma(num1: Int = 1, num2: Int = 2) : Int {`
`  return num1 + num2`
`}`
	`int soma([int numero1 = 1, int numero2 = 2]) {`
	`  return numero1 + numero2;`
	`}`
		Em Dart, usam-se [] para definir valores por omissão para parâmetros posicionais. Para *named parameters*, é ligeiramente diferente, ver abaixo

`fun soma(num1: Int, num2: Int) = num1 + num2`	`int soma(num1: Int, num2: Int) => num1 + num2;`	A sintaxe das funções "one-liner" é similar. Em Dart chamam-se *arrow functions*

`fun enableFlags(Boolean? bold, Boolean? hidden) {...}`
`...`
`enableFlags(hidden = false, bold = true)`
	`void enableFlags({bool? bold, bool? hidden}) {...}`
	`...`
	`enableFlags(hidden: false, bold: true);`
		Em Dart é possível definir *named parameters*, que são referenciados pelo nome em vez da posição (e por isso podemos mudar a ordem dos mesmos), usando as chavetas
		Em Kotlin, os parâmetros são automaticamente *named*

`fun soma(num1: Int = 1, num2: Int = 2) : Int {`
`  return num1 + num2`
`}`
	`int soma({int numero1 = 3, int numero2 = 4}) {`
	`  return numero1 + numero2;`
	`}`
		Com *named parameters*, é possível definir valores por omissão