Programació multimèdia
i dispositius mòbils

3. Variables i Operadors

IES Jaume II El Just
Tavernes de la Valldigna
Departament d'Informàtica
Curs 2021-22

Continguts

1 Variables i tipus de dades

1.1 Variables i tipus bàsics en Java

Els tipus de dades *primitius* o bàsics en Java són els següents:

- **Lògic**: `Boolean`
- **Caràcter**: `char`
- **Enter**: `byte`, `short`, `int`, `long`
- **Reals**: `float`, doube

Com sabem, aquests tipus tenen una classe de cobertura (*wrapper*), que serveix per tractar les dades primitives com objectes (`Integer`, `Float`...)

Per tal de declarar una variable en Java, feiem:

```
TipusBasic NomVariable;
```

En aquest cas, la variable estaria declarada, però no tindria cap contingut. Si accedim a aquesta variable sense assignar-li un valor obtindriem un error.

Si volem assignar-li valor directament, podem fer:

```
TipusBasic NomVariable=valorVariable;
```

Les variables es poden modificar al codi Java. Si volem que una variable siga immutable, és a dir, **per tal de definir una constant** farem ús de la paraula `final` al davant:

```
final float PI=3.14f;
```

1.2 Variables i tipus bàsics en Kotlin

En Kotlin els tipus de dades són classes, de manera que podem accedir a les seues *propietats i funcions membre* -Recordeu que amb Java per fer açò utilitzem *wrappers*-. Alguns d'aquests tipus, com els números, caràcters o valors lògics poden representar-se de forma especial internament, com a valors primitius en temps d'execució, però tot això de manera transparent a l'usuari.

Per definir variables en Kotlin fem ús de les paraules reservades `varo` `val`:

- Utilitzarem `var` per tal de definir **variables mutables**

- Utilitzarem `val` per tal de definir **variables immutables**, o el que en Java sería **valors constants**.

En general, es recomana utilitzar valors constants sempre que sapiguem que no van a ser modificats, per disposar de major seguretat i rendiment quan treballem en diversos fils d'execució.

Les següents assignacions serien correctes:

```
val pi = 3.14   // Constant

val modul       // Constant
modul = "PMDM"  // Encara que siga altra línia, és una única assignació

var x = 1       // Valor variable
x=x+1
```

Si ens fixem, veurem que **no hem definit el tipus de les variables**. Kotlin és capaç d'**inferir** el tipus de les variables a partir dels valors amb què les inicialitzem, pel que no cal indicar el tipus explícitament. Només serà necessari indicar el tipus d'una variable si no li donem valor en la declaració. Per tal d'indicar el tipus farem:

```
var nomVariable: Tipus
```

O bé podem indicar el tipus i donar també valor:

```
val nomVariable: Tipus = valor
```

Com hem dit, en Kotlin tots els tipus de dades són classes. Totes estes classes descendeixen de la superclasse genèrica Any, que podría equiparar-se a la classe `Object` de Java o C#.

1.2.1 Tipus numèrics

En Kotlin tenim els següents tipus numèrics:

Tipus	Exemple	Longitud	Conversió al tipus
Byte	`val byte: Byte = 8`	8 bits	`toByte(): Byte`

Tipus	Exemple	Longitud	Conversió al tipus
Short	`val short: Short = 16`	16 bits	`toShort(): Short`

Int	`val int: Int = 32`
	`val hexadecimal: Int = 0x16`
	`val binary: Int = 0b101`
		32 bits	`toInt(): Int`

! Long	`val long: Long = 64L`	64 bits	`toLong(): Long`
Foat	`val float: Float = 32.0F`	32 bits	`toFloat(): Float`
Double	`val double: Double = 64.0`	64 bits	`toDouble(): Double`

Com hem dit, la inferència dels tipus quan no els especifiquem, es fa en base als valors assignats, de manera que podem especificar de forma indirecta el tipus, per exemple:

```
val myVar=1 //Inferiria el tipus de myVar com a Int
```

```
val myVar=1L //Inferiria el tipus de myVar com a Long
```

```
val myVar=1.0 //Inferiria el tipus de myVar com a Double
```

```
val myVar=1.0F //Inferiria el tipus de myVar com a Float
```

Per altra banda, podem afegir guions baixos (_) quan escrivim números per tal que es puguen llegir millor, per exemple:

```
val milio = 1_000_000
```

1.2.2 Caràcters

El tipus caràcter a Kotlin es representa amb `Char`, i a diferència de Java, no es representa com un número.

Per definir un tipus caràcter fem:

```
var caracter:Char='a'
```

I per fer la conversió a caràcter, utilitzem:

```
toChar(): Char
```

Per exemple (tenint en compte que el codi ASCII del caràcter 'a' és el 97):

```
>>> 97.toChar()
res15: kotlin.Char = a
```

Per indicar caràcters especials, farem ús de seqüències d'escapament, mitjançant la barra invertida: `'\t','\b','\n','\r','\'','\"','\\','\$'`.

1.3 Cadenes de caràcters

A Java, les cadenes de caràcters es representen amb la classe `java.lang.String`.

Per declarar una cadena de caràcters, podem indicar-no directament o creant un instància de la classe *String*:

```
String cadena="contingut";
String cadena=new String("contingut");
```

En Kotlin la representació de cadenes de caràcters és molt semblant a Java. També s'utilitzen les cometes dobles, i podem escapar caràcters amb la barra invertida `\`.

```
val cadena="contingut"
val cadena:String="contingut"
```

Alguns dels mètodes interessants per a tractar amb cadenes de caràcters són, en Java i Kotlin:

Mètode de String en Java	Funció/Propietat membre en Kotlin
`cadena.concat(cadena2)`	`cadena1.plus(cadena2)`
`cadena.length()`	`cadena.length`
`cadena.charAt(pos)`	`cadena[pos]`
`cadena.substring(pos_ini[, pos_fy])`	`cadena.substring(pos_ini[, pos_fy])`
`cadena.equals(cadena2)`	`cadena.equals(cadena2)`
`cadena.startsWith(cadena2)`	`cadena.startsWith(cadena2)`

A més, per a la concatenació (*concat/plus*) i per a la comparació (*equals*), disposem dels operadors sobrecarregats + i == respectivament, tant en Java com en Kotlin. També podrem concatenar valors d'altres tipus, sempre que algun element siga una cadena (per exemple: `val cadena="hola"+1`).

1.3.1 Cadenes sense format o Raw String

Les cadenes sense format es delimiten amb triples cometes, i ens permeten escriure un *String* en diverses línies. Aquestes podran contenir tant noves línies com qualsevol altre caràcter:

```
val fragment="""
El miedo es el camino hacia el Lado Oscuro;
el miedo lleva a la ira,
la ira lleva al odio,
el odio lleva al sufrimiento.

        Maestro Yoda
                    ____
                 _.' :  `._
             .-.'`.  ;   .'`.-.
    __      / : ___\ ;  /___ ; \      __
  ,'_ ""--.:__;".-.";: :".-.":__;.--"" _`,
  :' `.t""--.. '<@.`;_  ',@>` ..--""j.' `;
       `:-..-J '-.-'L__ `-- ' L_..-;'
         "-.__ ;  .-"  "-.  : __.-"
             L ' /.------.\ ' J
              "-.   "--"   .-"
             __.l"-:_JL_;-";.__
"""
```

Amb aquestes cadenes podem utilitzar el mètode `trimMargin()` que elimina els espais en blanc abans del caràcter `|`. Açò ens pot ser d'utilitat quan volem mantindre la indentació a l'hora de definir una cadena sense format. Per exemple si fem:

```
fun main(args: Array<String>) {
  println("""El miedo es el camino hacia el Lado Oscuro;
            el miedo lleva a la ira,
            la ira lleva al odio,
            el odio lleva al sufrimiento.

                    Maestro Yoda
        """)
}
```

Obtindríem l'eixida:

```
El miedo es el camino hacia el Lado Oscuro;
            el miedo lleva a la ira,
            la ira lleva al odio,
            el odio lleva al sufrimiento.

                    Maestro Yoda
```

Per tal de mantenir la indentació i escriure el fragment correctament, fariem:

```
fun main(args: Array<String>) {
  println("""El miedo es el camino hacia el Lado Oscuro;
            |el miedo lleva a la ira,
            |la ira lleva al odio,
            |el odio lleva al sufrimiento.

            |        Maestro Yoda
        """.trimMargin())
}
```

El caràcter `|` amb el que podem indicar fins on eliminar espais inicials, pot ser reemplaçat per altre passant-lo com a argument al mètode. Per exemple, per a que siga `#`, fariem: `.trimMargin("#")`.

1.3.2 String templates

Els *String Templates* o literals de cadena poden contindre expressions de plantilla (*template expressions*): fragments de codi que serà avaluat i el seu resultat concatenat a la cadena. Amb això podem incloure valors, variables o fins i tot expressions dins una cadena.

Les expressions de plantilla comencen amb el signe dòlar $ i consisteixen en un nom de variable o una expressió entre claus {}.

A l'apartat sobre l'ús d'arguments ja hem vist una expressió d'aquest tipus:

```
println("Hello ${args[0]}")
```

Però podem utilitzar altres construccions com:

```
val nom="maestro Yoda"
println("$nom te ${nom.length} caràcters")
```

```
val temperatura = 27
println("Amb ${temperatura}º fa ${if (temperatura > 24) "calor" else
 ↪  "fred"}")
```

Per mostrar la variable `temperatura` podriem haver utilitzat `$temperatura` sense claus, però com hem afegir el símbol º al darrere, hem hagut d'utilitzar les claus per delimitar el nom de la variable.

Per altra banda, aquestes plantilles es poden utilitzar dins de qualsevol tipus de cadena, tant sense processar com escapades, pel que si volem escriure *literalment* una expressió fariem (representem $ amb `${'$'}`):

```
val temperatura=27
println("""El valor de ${'$'}{temperatura} és ${temperatura}º""")
```

1.4 Valors nuls o *Nullable Types*

Hem comentat que Kotlin és un llenguatge segur, i entre altres coses, ens evita errors a l'hora de programar com el *NullPointerException* ja que no permet que els valors de les variables siguen nuls per defecte.

Si volem especificar que una variable puga contindre un valor `nul`, cal definir-la explícitament com a *nullable*. Per a això, quan la definim, afegim un interrogant ? al seu tipus:

```
>>> val nom: String?=null
```

Si no haverem definit la variable com a *nullable* haverem obtingut l'error:

```
>>> val nom:String=null
error: null can not be a value of a non-null type String
```

1.4.1 L'operador *Safe Call Operator* ?.

Per tindre un plus de seguretat a les variables *nullables* i poder d'accedir a les propietats o mètodes d'un objecte d'aquest tipus, Kotlin ens ofereix l'operador *Safe Call Operator* `?.`. Amb aquest operador, només podrem accedir a atributs o mètodes d'un objecte si té un valor no nul. En cas que aquest siga nul, s'ignorarà, evitant així una excepció de tipus `NullPointerException`. Veiem alguns exemples:

- Definició d'un string *nullable* amb valor **diferent** a `nul`, i accés a la seua propietat `length`, que ens diu la longitud d'aquest (com que la variable és *nullable*, hem d'utilitzar necessàriament l'operador segur `?.`):

```
>>> val nom:String?="Jose"
>>> nom?.length
res1: kotlin.Int? = 4
```

- Definició d'un string *nullable* amb valor **igual** a `null`, i accés a la seua propietat `length`, que ens retornarà `null`, però sense tornar cap error.

```
>>> val nom:String?=null
>>> nom?.length
re24: kotlin.Int? = null
```

- En canvi, si definim un string sense indicar que siga nullable, si li assignem el valor `null` obtindrem un error:

```
>>> val nom:String=null
error: null can not be a value of a non-null type String
```

Per altra banda, podem encadenar *crides segures*, fent ús de l'operador `?.`. Per exemple:

```
// La funció ObtenirNomPaisSegur retorna un String corresponent
// al país en què viu la persona (objecte de tipus Prsona)
// que li passem com a argument.
fun ObtenirNomPaisSegur(persona: Persona?): String? {
    // Obtenim el nom del país on viu la pesona "navegant"
    // a través de les diferents classes
    return persona?.direccio?.ciutat?.pais?.nom
}
```

Amb l'ús de l'operador `?.` ens estem assegurant que si algun dels objectes intermitjos és *nul*, el valor de retorn de la funció siga *nul* i no llance cap error.

1.4.2 L'operador *Elvis* `?:`

Kotlin també ens proporciona l'operador `?:`, conegut com l'operador *Elvis*, per tal d'especificar un valor alternatiu quan la variable és nul·la. Veiem un exemple d'ús des de la consola de kotlin:

```
>>> val nom:String?="Jose"
>>> nom?.length ?: -1
res39: kotlin.Int = 4

>>> val nom:String?=null
>>> nom?.length ?: -1
res41: kotlin.Int = -1
```

En aquest exemple, si la cadena *nom* té valor, quan accedim a la seua longitud, ens mostrarà aquesta sense problemes. En canvi, si *nom* no té valor (aquest és `null`), obtindrem el valor de *-1*, en lloc d'un error.

1.4.3 Operador d'aserció no-nul `!!`

Aquest operador `!!` convertirà qualsevol valor de tipus *nullable* a un tipus **no nul**, i tornarà una excepció *NullPointerException* en cas que el valor siga nul.

- Definim un string *nullable*, amb valor nul:

```
>>> val nom1:String?=null
```

- Si accedim a la seua propietat length obtindrem un error:

```
>>> nom1.length
error: only safe (?.) or non-null asserted (!!.) calls are allowed on a
 ↪  nullable receiver of type String?
nom1.length
    ^
```

Aquest error ens indica que estem intentant accedir a les propietats d'un objecte que hem definit com a *nullable* com si aquest fora segur (no *nul·lable*). Per evitar l'error, ens hem d'utilitzar bé l'operador de crida segura (`?.`) o convertim la variable a un tipus no nul amb l'operador d'asserció no nul (`!!`).

- Utilitzant aquest operador d'asserció no nul, veurem com ara Kotlin ens torna una excepció:

```
>>> nom1!!.length
kotlin.KotlinNullPointerException
```

La utilitat d'aquest operador és per forçar que es produisca una excepció `NullPointerException` quan accedim a un valor nul.

Podeu trobar més informació sobre els tipus segurs a la documentació de Kotlin: https://kotlinlang.org/docs/reference
safety.html

1.5 Detalls sobre l'emmagatzemament de variables

L'emmagatzemament de variable en **Java** es realitza de la següent forma:

- Les variables de ***tipus bàsic*** (int, float,…) s'emmagatzemen tal qual en l'adreça de memòria a la que apunta la variable.
- Per a les variables de ***tipus no bàsic*** (objectes, vectors i matrius), l'adreça de memòria a la que apunta el nom de la variable és una referència a l'adreça de memòria on realment es guarda l'objecte.

En **Kotlin** el mecanisme és pràcticament igual, tenint en compte que els tipus bàsics s'implementen també com a objectes.

Amb açò, anem a tindre algunes consideracions sobre el pas de paràmetres i la comparació d'objectes:

- Quan s'invoca un mètode en Java, el pas de paràmetres és sempre **per valor.** Ara bé:
  - Quan passem una variable de tipus bàsic, es fa una còpia del valor d'aquesta, i si es modifica, no afecta a la variable real.

- Quan passem una variable de tipus no bàsic, es fa una còpia, però de la referència a l'objecte, pel que els canvis sí que afecten a l'objecte original.

A l'hora de **comparar objectes** en Java, si utilitzàrem l'operador ==, compararíem la referència en memòria, el que voldria dir que comprovariem si és el mateix objecte, no si té el mateix valor. Per tal de fer la comparació d'objectes, **caldría implementar un mètode específic** en la pròpia classe que comprovara que es tracta d'objectes de la mateixa classe, i comparara un per un tots els atributs d'aquest.

En Kotlin, tot i que els tipus bàsics s'implementen també com a objectes, i per tant s'emmagatzemen les seues referències, es tracten d'una forma més intel·ligent, i podem utilitzar els operadors de comparació de la mateixa manera que en Java. Al següent apartat ho veurem amb més detall.

2 Operadors

Als següent apartats, veurem les taules amb els diferents operadors que podem utilitzar tant en Java com en Kotlin i el seu significat.

- **Operador d'assignació**

S'utilitza per donar valor a una variable.

Operador	Significat	Exemple
=	Assignació	n=4

- **Operadors aritmètics**

Serveixen per realitzar operacions aritmètiques amb les variables. El resultat serà un valor numèric.

Operador	Significat	Exemple
-	Canvi de signe (unari)	-4
+	Suma	5 + 3
++	Increment	5++
+=	Suma Combinada	a+=b (a=a+b)
-	Resta	5 - 3
–	Decrement	5–

Operador	Significat	Exemple
-=	Resta Combinada	a-=b (a=a-b)
*	Producte	5 * 3
*=	Producte Combinat	a=*b (a=a*b)
/	Divisió	5/3 5.0/3.0
/=	Divisió Combinada	a/=b (a=a/n)
%	Resta de la divisió entera	5 % 3
%=	Resta de la divisió combinada	a%=b (a=a%b)

- **Operadors relacionals**

Serveixen per realitzar comparacions entre variables, i retornen un valor lògic.

Operador	Significat	Exemple
==	igual que	a==b
!=	diferent que	a!=b
<	menor que	a<b
>	major que	a>b
<=	menor o igual	a<=b
>=	major o igual	a>=b

A més, amb ***Kotlin***, podem utilitzar:

Operador	Significat	Exemple
===	És el mateix objecte	a===b
!==	No és el mateix objecte	a!==b

En Kotlin el diferencia la igualtat *estructural* i la igualtat *referencial*:

- La igualtat **estructural** serveix per tal de comprovar si dos valors o variables són iguals (com un `equals()` en Java), i utilitza els operadors `==` per comprovar si tenen el mateix valor i `!=` per

comprovar si són valors diferents.
- La igualtat **referencial**, per la seua banda, comprova si dues referències apunten o no al mateix objecte. Amb === comprovem si s'apunta al mateix objecte, i amb !== si dos referències apunten a objectes diferents.

Per altra banda, tot i que a Kotlin tot són objectes (per tant, referències) amb els tipus bàsics (int, float…) es fa una excepció, i es tracten de manera més intel·ligent, de manera que la igualtat referencial === compara valors en lloc de comparar les referències.

- **Operadors lògics**

Serveixen per realitzar operacions entre variables de tipus lògic. El seu resultat és també de tipus lògic.

Operador	Significat	Exemple
!	Negació	!(a==b)
||	OR	(a==b)||(a==c)
&&	AND	(a==b)&&(a==c)

- **Operador condicional ternari**

En Java existeix l'operador condicional ternari:

Operador	Significat	Exemple
? :	Operador Condicional Ternari	a= (b==0 ? a : a/b)

```
valor = ( condició ? expressió_1 : expressió_1);
```

Que sería equivalent a:

```
if (condició)
    valor = expressió_1;
else
    valor = expressió_2;
```

Cal dir, que en aquest punt, Java, a l'igual que altres llenguatges com Javscript o C#, tracta les sentències condicionals com una declaració, el que vol dir que no es resolen a un valor. Per això, cal utilitzar

aquest operador ternari si volem assignar un valor de forma condicional a una variable.

En Kotin, una sentència condicional no és una declaració, sinò una expressió, de manera que pot assignar-se directament a una variable, de manera que no és necessari aquest operador, i podem fer coses com:

```
var valor = if (condició) expressió_1 else expressió_2
```

Veiem-ho amb un exemple més clar:

```
// En Java:
major = ( x>y ? x : y);
```

```
// En Kotlin:
var major = if (x>y) x else y
```

Com veiem, resulta una miqueta més clar el codi en Kotlin que en Java.

- **Separadors**

Els separadors són caràcters amb significat especial.

Operador	Significat
()	Permet especificar la prioritat dins les expressions i fer conversions de tipus (Java). També s'usa per especificar la llista d'arguments de funcions i/o mètodes
{}	Defineix blocs de codi
[]	Per declarar i referenciar elements de vectors o matrius
;	Separador de sentències (opcional en Kotlin)
,	Separa identificadors en la declaració de variables i llistes de paràmetres, o encadenar sentències dins un for
.	Separa el nom d'un atribut/mètode de la seua instància de referència. També separa l'identificador d'un paquet dels subpaques i classes