Apunts

Descarregar aquests apunts

Tema 0 . Programació Bàsica en Kotlin

Índex

Introducció

Kotlin és un llenguatge de programació fortament tipat desenvolupat per **JetBrains** en 2010 i que està influenciat per llenguatges com **Scala , Groovy i C#**.
La majoria de desenvolupaments amb Kotlin, tenen com a destí Android o la màquina virtual de java (JVM), i encara que també ofereix la possibilitat de desenvolupaments per a la màquina virtual de Javascript o codi natiu, ara com ara encara són pocs els programadors que ho usen per a aquests destins.
A partir de l'actualització Kotlin 1.3.30, es van incloure les millores per a Kotlin/Native que permet compilar el codi font de Kotlin en dades binàries independents per a diferents sistemes operatius i arquitectures de CPU, inclòs IOS, Linux, Windows i Mac.

En el següent enllaç pots trobar un compilador de Kotlin, i altres llenguatges, en línia que permet entrada de dades per consola, el pots usar per a provar els exemples.

Control de Nuls

Kotlin ha aportat una sèrie d'elements que permeten realitzar un major control dels tipus que poden ser nuls Null safety, i que pretén evitar la tan coneguda excepció **NullPointerException**.
Com ocorre amb altres llenguatges, Kotlin permet definir un tipus no nul com a anul·lable. Això ho fa mitjançant l'element `?`, per exemple una variable de tipus sencera no podria emmagatzemar un valor nul, però es pot canviar aquesta condició de la següent manera.

```
fun main()
{
   var numero:Int
   numero=null //ERROR de compilació, els tipus valor no són anul·lables
   var numeroAnulable:Int?
   numeroAnulable=null //OK
}
```

Per a comprovar si les variables són nul·les, sempre es pot usar el condicional if/else, encara que Kotlin ens permet altres opcions. L'operador **anomenada segura** `?.` que només realitzarà la crida en cas que el valor siga diferent de nul i retornarà null en un altre cas.

```
fun main()
{
   var cadenaAnulable:String?=null
   println(cadenaAnulable?.length)
}
```

En aquest exemple, si no usàrem l'operador de crida segura, el codi llançaria NPE. Però ara mostrarà **null**.

Un altre operador, no tan recomanat és l'operador de **asserció no nul·la** `!!`. converteix qualsevol valor en un tipus no nul i llança una excepció si el valor és null. Podem dir d'aquest operador, que torna les coses a la normalitat ja coneguda.
Per tant, si desitges un NPE, pots tindre-la sol·licitant-la amb aquest operador.

```
fun main()
{
   var cadenaAnulable:String?=null
   println(cadenaAnulable.length) //ERROR de compilació, no permet possibilitat de produ
   println(cadenaAnulable!!.length) //Sol·licitem aquesta opció amb l'operador !!
}
```

I finalment tenim l'operador **elvis** `?:` que comprova que el valor no és nul, permetent la crida en aqueix cas o retornant el que *decidim* en cas que siga nul.

```
fun main()
{
   var cadenaAnulable:String?=null
   println(cadenaAnulable?.length?:0)
}
```

Si l'expressió a l'esquerra de ?:no és null, es realitza l'anomenada a length, en cas contrari retorna l'expressió de la dreta, en aquest cas 0.

Variables

Igual que en tots els llenguatges de programació, en Kotlin també tindrem el recurs de les variables per a emmagatzemar valors. En Kotlin ens podem trobar amb variables **immutables** i variables **mutables**. En el cas de les primeres, una vegada se li assigna un valor a la variable no podrà ser modificat, és a dir, es comporta com una constant (el mateix que utilitzar final a Java o const en C#). Mentre que amb la mutables podrem modificar en qualsevol moment el valor de la variable.
Per a declarar una variable com a mutable, l'haurem de precedir de la paraula clau **var** mentre que per a les immutables usarem **val**.
El concepte d'inmutabilitat és molt interessant. Al no poder canviar els objectes aquests són més robustos i eficients. Sempre que puguem hem d'usar objectes immutables.

```
//Variables mutables
var mutable:Int=5;
mutable+=7;
var numeroDecimales=3.14F;
numeroDecimales=3.12F;
//Variables inmutables
val inmutable:Char='C';
inmutable='A'   //Error compilació!! no es pot modificar una variable immutable
```

Com es pot veure en l'exemple, quan declarem una variable, podem indicar el tipus d'aquesta o esperar que el compilador l'inferisca.

Per a definir el tipus, haurem d'indicar-lo amb `:tipus` després del nom:

(var|val) nombreVariable [:tipus][=valor]

Els tipus bàsics de variables en Kotlin són:

Tipus	Valor	Grandària
Byte	5.toByte()	8 bits
Short	5.toShort()	16 bits
Int	5	32 bits
Long	5L	64 bits
Float	1.45F	32 bits
Double	1.45	64 bits
Boolean	true	
Char	'H'	16 bits
Unit	Unit	0 bits

> El tipus `Unit` correspon a void a Java i C#

Quan una **variable mutable** declarada en el cos d'una classe, no es vol inicialitzar en el moment de la declaració, existeix el concepte de `Inicialització tardana`, afegint el modificador **lateinit** davant de la declaració de la variable.

```
public class Ejemplo{
lateinit var cadena: String
fun miFuncion() {
  cadena = "Hola Mon";
  println(cadena);}
}
```

> ✋ Com en altres llenguatges de programació, s'ha de controlar les operacions amb variables de diferents tipus, per a evitar resultats inesperats o fins i tot excepcions. Kotlin té diversos mètodes **toX()** per a canviar els valors al tipus que necessites:

```
fun main (args: Array <String>) {
   var a:Int
   var b=3.5f
   a=b+2
   println(a);
}
```

> Produiria l'error ***error: type mismatch: inferred type is Float but Int was expected.***
> Mentre que: `a=b.*toInt()+2` , evitaria l'error, encara que la funció arredoniria a 3.

String

El tipus String representa el literal de cadena ja conegut, podem veure un exemple de l'ús de templates `$` per a l'eixida per pantalla.

```
fun main (args: Array <String>) {
val cadena = "El resultat de:"
val a = 2
val b = 5
println("$cadena $a + $b és: ${a + b}")
}
```

```
El resultat de: 2 + 5 és: 7
```

Control de flux

El control de flux en kotlin té algunes diferències interessants a altres llenguatges més coneguts. Les instruccions **if/else, for i while** es poden considerar similars, així que explicarem solament els elements que les diferencien:

for y while

Es diferencia sobretot en el fet que s'han d'usar rangs en la sentència for i es poden usar en la while:

```
fun repeticioAmbFor(v:Int)
{
   for(i in (0..v).reversed()) println("$i")
}
fun main() { repeticioAmbFor(10)}
```

```
fun repeticionAmbWhile(tope:Int)
{
   var contador:Int=0
   do{
         var numero=readLine()!!.toInt();
         contador++;
   }while(contador<tope && numero!in 1..100)
}
fun main() { repeticionAmbWhile(10)}
```

when

Substitut de **swich**, i que també ho podem trobar similar en C#. Amb un exemple quedaria explicat:

```
fun getEstacio(entrada:Int):String
{
return when(entrada)
{
    1-> "primavera"
    2-> "estiu"
    3-> "tardor"
    4-> "hivern"
    else -> "Estació incorrecta"
}
}
fun main() { println(getEstacio(2))}
```

Un altre exemple de la gran funcionalitat que ens permet la sentència **when** podria ser el següent. Observa que en aquest cas el when és **sense arguments**:

```
fun noFaigRes(x:Int, s:String, v:Float):String
{
   val res = when {
      x in 1..10 -> "enter positiu menor de 10"
       s.contains("cadena") -> "incloc cadena"
       v is Comparable<*>-> "Soc Comparable"
       else -> ""
   }
   return res
}
fun main() { println(noFaigRes(2, "Hola", 3.5f));}
```

Funcions

Si queremos crear una función en kotlin tendremos que precederla de la palabra reservada **fun**. Por tanto, una función constará de la palabra fun seguida por el nombre de la función y entre paréntesis los parámetros, siempre y cuando los tenga. Si la función retorna un valor se definirá al final de la signatura (esto último se puede omitir siempre que la función no devuelva nada).
Si volem crear una funció en kotlin haurem de precedir-la de la paraula reservada **fun**. Per tant, una funció constarà de la paraula fun seguida pel nom de la funció i entre parèntesi els paràmetres, sempre que els tinga. Si la funció retorna un valor es definirà al final de la signatura (això últim es pot ometre sempre que la funció no retorne res).

fun nomFuncio([nomVariable:Tipus, nomVariable:Tipus, ...]):TipusDevolució

```
fun sumaDades(datoUno:Int, datoDos:Char )
{
   println("${datoUno + datoDos.toInt()}");
}
fun main() {
   var mutable:Int=5;
   mutable+=7;
   var numeroDecimals=3.14F;
   numeroDecimals=3.12F;
   val inmutable:Char='C';
   sumaDades(mutable,inmutable);
}
```

Amb valor de retorn:

```
fun major(numeroUno:Int, numeroDos:Int):Int
{
 return if(numeroUno>numeroDos)  numeroUno else numeroDos
}
fun main() { println(major(5, 7));}
```

Si la funció només té una instrucció, com en el cas anterior, Kotlin permet ometre les claus.

```
fun major(numeroUno:Int, numeroDos:Int):Int= if(numeroUno>numeroDos)  numeroUno else num
```

Funcions de Extensió

Kotlin igual que altres llenguatges, ens permet estendre la funcionalitat de classes existents, agregant funcions a aquestes. En el següent exemple, s'ha afegit la funció Escriu() a la classe

String, per la qual cosa es podrà usar amb objectes d'aquest tipus.

```
String.Escriu()=println(this);
main() {"Soc una cadena".Escriu()}
```

Classes i Interfícies

Tots els elements en Kotlin són públics per defecte, per la qual cosa també ho seran les classes. Les classes ens serveixen per a referenciar objectes del món real, i mitjançant les propietats podem definir les diferents característiques que ens interesse manipular d'aquests. En Kotlin ja no existeixen els atributs, sinó que tot passa a ser propietats, similars a les que utilitza C#, se'ls pot donar funcionalitat si ho necessitem.

Una classe Persona amb dues propietats nom i edat a les quals volem donar funcionalitat, podria quedar així:

```
class ExcepcioEdat(cadena:String):Exception(cadena){}
class Persona
{
     val nom:String
     get() = field.toUpperCase()
     var edat: Int=0
       set(value)
       {
           if(value <0 || value>125) throw ExcepcioEdat("Edat Invàlida")
           else field = value
       }
}
```

Si et fixes, Kotlin utilitza la paraula clau `field` per a referir-se a la propietat en si, per la qual cosa no ha de crear variables secundàries com ocorre amb C#.

Constructors

Les classes poden tindre un constructor principal i un o més constructors secundaris. El constructor principal és part de l'encapçalat de la classe, això ens permet construir un objecte sense haver de definir el constructor. Per exemple, per a definir la classe Persona anterior però sense funcionalitat en les propietats, només bastaria amb fer el següent:

```
class Persona(val nom: String, val edat: Int=0)
{

}
```

I la creació de l'objecte seria:

```
val persona=Persona("Pepe", 23)
```

Si necessitem que ocórrega alguna cosa en el moment de crear l'objecte, es pot usar la clausula `init` que serà executada en cridar al constructor principal. Si et fixes en el següent codi, la propietat edat està declarada en el constructor, mentre que el nom s'ha declarat en el codi de la classe.

```
class Persona(nom:String, var edat:Int=0)
{
     val nom:String
     get() {
         return field.toUpperCase()
     }
   init{
       this.nom=nom
       if(edat<0 || edat>125)  throw ExcepcioEdat("Edat Invàlida")
       else this.edat=edad
   }
}
```

Si lo que queremos es crear más de un constructor, tendremos que recurrir a la clausula `constructor` indicando los parámetros necesarios y llamando al constructor por defecto con `this` para enviarle sus propiedades. Ahora hemos añadido una propiedad más a la clase Persona, el dni que será asignado con el segundo constructor
Si el que volem és crear més d'un constructor, haurem de recórrer a la clausula `constructor` indicant els paràmetres necessaris i cridant al constructor per defecte amb `this` per a enviar-li les seues propietats. Ara hem afegit una propietat més a la classe Persona, el dni que serà assignat amb el segon constructor

```
class Persona(nom:String, var edat:Int=0)
{
   var dni:String="CAP"
   val nom:String
   get()=field.toUpperCase()
   init{
       this.nom=nom
       if(edat<0 || edat>125)  throw ExcepcioEdat("Edat Invàlida")
       else this.edat=edat
   }
   constructor(nom:String, edat:Int=0, dni:String):this(nom, edat)
   {
       this.dni=dni
   }
}
```

Herència

En Kotlin, la classe Any és l'arrel de la jerarquia de classes. Cada classe en el llenguatge derivarà d'ella si no especifiques una superclase. Seria similar a la classe Object de C# i Java.

D'altra banda en Kotlin, tant les classes com els membres d'aquestes són tancats, això significa que no es pot heretar d'una classe i tampoc es poden sobreescriure els seus membres si no ho indiquem explícitament. Perquè d'una classe es puga heretar caldrà afegir-li el modificador `open`. Per exemple, si la classe Persona volem fer-la oberta seria:

```
open class Persona(nom:String, var edat:Int=0)
{
  ...
}
```

I ara podríem crear una classe filla de Persona, com per exemple:

```
class Estudiant(nom:String,  edat:Int=0, var estudis:String):Persona(nom, edat)
{

}
```

Suposant que la propietat dni volem fer-la que es puga reescriure i que té un nou mètode també volem que es puga reescriure anomenat imprimir i un normal esMajor, ara la classe quedaria:

```
open class Persona(nom:String, var edat:Int=0)
{
   open var dni:String="CAP"
   ...
   open fun imprimir()= println("Nom: $nom  Edat: $edat")
   fun esMajor() = if (edat >= 18) true else false
}
```

I amb els elements que volem reescriure en la classe Estudiant:

```
class Estudiant(nom:String,  edat:Int=0, var estudis:String):Persona(nom, edat)
{
   override var dni:String="ESTUDIANT SENSE DNI"
   override fun imprimir()
   {
       super.imprimir();
       println("Soc estudiant de $estudis")
   }
}
```

Interfícies

En Kotlin podem implementar classes abstractes, que són iguals a les que ja coneixem d'altres llenguatges, per la qual cosa no comentarem res sobre elles. A més també es poden crear interfícies, que permeten definir tipus els comportaments dels quals poden ser compartits per diverses classes que no estan relacionades. Usa la paraula reservada `interface` i la seua implementació és similar als llenguatges que coneixem amb algunes xicotetes diferències. Com ja sabem permeten l'herència múltiple, i a més:

- Poden contindre mètodes abstractes (sense implementació) i mètodes regulars (amb implementació)
- Pot contindre propietats abstractes i regulars
- No permet declaració de constructors
- Les propietats i mètodes regulars d'una interfície poden ser sobreescrits amb el modificador override sense haver de marcar-los amb open, a diferència d'en les classes abstractes.

```
interface IEstudis
{
   var curs: Int // Propiedat abstracta
   val ultimCurs: Boolean // Propiedat regular
       get() = curs ==2
   fun estudis():String // Mètode abstracte
   fun socEstudiant() { // Mètode regular
    println("Soc Estudiant de " +estudis())
}
}
```

I ara fem que la classe Estudiant a més d'heretar de Persona, implemente la interfície IEstudis, quedant:

```
class Estudiant(nom:String,  edat:Int=0, >var estudis:String):Persona(nom, edat), >IEstu
{
   override var curs:Int=0
    set(value) {field=curs}
   override var dni:String="ESTUDIANT SENSE >DNI"
   override fun estudis()= estudis
}
```

Classes Enum en Kotlin

Igual que en altres llenguatges, Kotlin ens permet crear tipus enumerats, encara que en aquest cas es pot veure com un modificador de classe. Una enumeració és un conjunt de valors que usen com a identificador un nom. Aquest nom es comporta com una constant en el nostre llenguatge. En marcar una classe amb el modificador `enum`, la declara com una d'enumeració.

```
enum class CiclesInformatica {SMX, ASIX, DAM, DAW}
fun nivellCicle(cicle:CiclesInformatica  ):String
{
    return when(ciclo)
    {
        CiclesInformatica.SMX -> "Mitjà"
        else ->  "Superior"
    }
}
fun main()
{
  var ciclo=CiclosInformatica.ASIX
  println(nivellCicle(cicle)) //Superior
}
```

Valor en les enumeracions

A més a les enumeracions en Kotlin també podem assignar-los un o més valors. Això es farà a través del constructor de la classe, ho podem veure en el següent exemple, en el qual al constructor se li ha afegit tant un valor sencer com un grau de tipus cadena.

```
    enum class CiclesInformatica( val valor: Int, val grau: String)
    {
      SMX(1,"Grau Mitjà"),
      ASIX(2,"Grau Superior"),
      DAM(3,"Grau Superior"),
      DAW(4,"Grau Superior")
    }
    fun main()
    {
10    for(v in CiclesInformatica.values())  println(v.name+" "+ v.ordinal)
11    CiclesInformatica.values().forEach{println("${it.valor} - ${it.name} - ${it.grau}")}
    }
```

```
SMX 0
ASIX 1
DAM 2
DAW 3
1 - SMX - Grau Mitjà
2 - ASIX - Grau Superior
3 - DAM - Grau Superior
4 - DAW - Grau Superior
```

> :*pushpin: L'eixida per pantalla mostraria primer els valors assignats a les propietats per defecte `name i ordinal` **Línia 10** i després una llista amb les propietats que s'han creat en el constructor `valor i estat` **Línia 11**.

Enumeracions amb comportament

També se'ls pot afegir un comportament a través de funcions abstractes o no, o fins i tot d'interfícies. En el següent exemple podem veure que al constructor se li ha afegit un elements més amb el nom complet del cicle, i a més tenim el comportament afegit mitjançant el mètode `informacioCompleta`. De manera que l'execució del programa ens traurà les sigles del cicle, el nom complet i el grau que li correspon a cadascun dels elements de l'enumeració.

```
enum class CiclesInformatica( val valor: Int, val grau: String, val nom : String)
{
  SMX(1,"Grau Mitjà","Sistemes Microinformàtics i Xarxes"),
  ASIX(2,"Grau Superior","Administració de Sistemes Informàtics en Xarxa"),
  DAM(3,"Grau Superior","Desenrotllament d'Aplicacions Multiplataforma"),
  DAW(4,"Grau Superior","Desenrotllament d'Aplicacions Web");

  fun informacioCompleta()= "${name} - ${nom} - ${grau}"
}
fun main()
{
  CiclesInformatica.values().forEach{println("${it.informacioCompleta()}")}
}
```

Classes i mètodes parametritzats

Kotlin també permet crear classes i mètodes amb algun dels seus membres de tipus genèric. La llista de paràmetres per a tipus s'inclouen en parèntesis angulars i se separen per coma si són diversos `<T, O, V,..>`

Classes Genèriques

Per a crear una classe amb un tipus parametritzat de manera que una de les seues propietats siga d'aqueix tipus, es farà de la següent manera:

```
class ClasseGenerica<T>(t: T, c:String)
{
  var t:T
  val c:String
  init
  {
    this.t=t
    this.c=c
  }
  override fun toString()= "${t} ${c}"
  fun metode(param: T) {t=param}
}
```

I si volguérem crear un objecte d'aqueixa classe amb la propietat parametritzada a tipus sencer, es podria fer `val objecte=ClasseGenerica(3, "Hola")`

Si volguérem realitzar una restricció del tipus parametritzat a la interfície Comparable, s'hauria de fer de la següent manera:

```
class ClasseGenerica<T: Comparable<T>>(t: T, c:String)
{
 ...
}
```

Funcions Genèriques

Per a les funcions genèriques el paràmetre de tipus s'afegirà just després de la paraula fun i les restriccions es faran de la mateixa manera que en les classes.

```
fun <T> funcioGenerica(param: T): T {
    return param
}
```

Col·leccions

Llistes

En Kotlin hi ha diferents maneres de generar **llistes d'objectes**: arrays, llistes immutables, llistes mutables, etc. Atenció, cal distingir la inmutabilitat de la col·lecció de la inmutabilitat de la variable

que la conté. En aquesta documentació només comentarem els array i les llistes mutables, la resta de col·leccions seran consultades si és necessari el seu ús.

> ✋ Es recomana aprofitar les característiques d'aquest llenguatge i crear el tipus de col·lecció apropiada per a les necessitats de cada moment, així aconseguirem major eficiència en les aplicacions.

	arrayOf	**listOf**	**arrayListOf**	**linkedListOf**
Descripció	Array de objetos tradicional	Lista inmutable	Lista mutable	Lista enlazada mutable
Literals	arrayOf(1, 2, 3)	listOf(1,2, 3)	arrayListOf(1, 2, 3)	linkedListOf(1, 2, 3)
Inmutabilitat	no	sí	no	no
Modificar longitud	no	no	sí	sí

Arrays

Els arrays són mutable, és a dir poden canviar el valor dels seus elements durant l'execució, però com ja coneixem per un altre llenguatges, la seua grandària no es pot canviar una vegada estiga definit, existint un altre tipus de col·leccions per a aquest ús.

> ✋ En Kotlin les declaracions i accés a les posicions del array no solen realitzar-se mitjançant claudàtors (*corchetes*), com en altres llenguatges, sinó que s'utilitzen els mètodes de la classe Array.

En el següent exemple podem veure diferents maneres de recórrer el array mitjançant un bucle for.

```
fun main()
{
   val weekDays = arrayOf("primavera", "estiu", "tardor", "hivern")
   for (dato in weekDays)  println(dato) //Similar al foreach de C#
   for (i in weekDays.indices)  println(weekDays.get(i))  //Accés mitjançant índex
   for ((posicion, valor) in weekDays.withIndex()) //Bolcant totes dues dades en una *tu
       println(""La posició $posicion conté el valor $valor")
}
```

A més Kotlin distingeix els arrays de primitives usant classes pròpies:
`BooleanArray, ByteArray, CharArray, ShortArray, IntArray, FloatArray, DoubleArray`, i la manera

de construir literals és mitjançant: booleanArrayOf, byteArrayOf, charArrayOf, shortArrayOf, intArrayOf, floatArrayOf, doubleArrayOf.

En el següent exemple s'ha creat un array de float de grandària 4, l'accés està realitzat mitjançant `[]` a més l'última línia produeix excepció en intentar realitzar un accés no permés.

```
fun main()
{
   val arrayFloat=FloatArray(4)
   arrayFloat[3]=2.5f;
   arrayFloat.set(0, 3f);
   print(arrayFloat[0])
   arrayFloat[4]=8.9f//Excepció per fora d'indexe
}
```

Si volem crear un array d'una mena d'objecte determinat, suposarem un tipus Persona amb el següent codi:

```
class Persona(val nom: String, val edat: Int)
{
   fun imprimir()= println("Nom: $nom  Edat: $edat")
   fun esMajor() = if (edat >= 18) true else false
}
```

El array de Persones ho podrem inicialitzar en el moment de creació del array, de la següent manera (en aquest cas només tindrà un element):

```
fun main()
{
   val personas=arrayOf(Persona("Ana",12))
   personas.get(0).imprimir()
}
```

Una altra opció és permetre que el array puga tindre valors nuls, per a afegir posteriorment els elements. En aquest cas haurem d'utilitzar l'operador de crides segures `?.`, que només cridarà al mètode en el cas que el valor no siga nul, evitant `NullPointerException`

```
fun main()
{
   val personas=arrayOfNulls<Persona>(2)
   personas.set(0,Persona("Ana",12))
   personas.get(0)?.imprimir()
}
```

Array Bidimensional

En kotlin els arrays de més d'una dimensió es tracten com arrays de arrays (similar a les taules dentades de C#). Pel que es poden crear files de diferents grandàries. El següent codi crea una matriu d'enters de 4 x 4 inicialitzada a valor 0 i en l'element segona fila i segona columna a valor 3.

```
fun main()
{
   val matriz = Array(4) { IntArray(4) }
   matriz[1][1]=3
   for (e in matriz)  println(e.contentToString())
}
```

Si no volem inicialitzar cadascuna de les files quan creguem el array, es pot fer posteriorment si es crear la matriu anul·lable. En aquest exemple creem una matriu de tres files, cadascuna d'elles amb grandària de columnes diferent (2, 3 i 4 respectivament) i inicialitzem tota la matriu a valor 3.

```
fun main()
{
   val dentada = arrayOfNulls<IntArray>(3)
   dentada[0] = IntArray(2)
   dentada[1] = IntArray(3)
   dentada[2] = IntArray(4)

   for(i in 0..dentada.size-1)
   for(j in 0..dentada[i]!!.size-1) dentada[i]?.set(j, 3)
}
```

Amb la següent línia de codi creem una matriu quadrada de grandària 4 amb la diagonal a 1.

```
val matriz = Array(4) { f -> IntArray(4) { c -> if(f==c)1 else 0 } }
```

Llistes Mutables

Les llistes mutables estan representades per la classe `ArrayList` que implementa la interfície MutableList. Segurament seran les que més s'utilitzen, perquè permeten un comportament més flexible. Però com s'ha comentat abans, per a millorar l'eficiència s'ha d'utilitzar la col·lecció més apropiada per a cada necessitat.

```
fun main()
{
   val persones=ArrayList<Persona>()
   persones.add(Persona("Ana",12))
   personas.add(Persona("Pedro", 15))
   for(p in persones) p.imprimir()
}
```

Per a accedir a un element s'usa la funció **get(*posició*)** i per a reemplaçar **set(*posició*, *element*)**, com en els arrays. Es disposa d'una funció que afig una col·lecció a la llista mutable ***addAll(coleccion)**, per a eliminar un element de la llista s'usarà **remove(*element*)** o **removeAt(*posició*)**. Per a buscar tenim **indexOf(*element*)** o **lastIndexOf(*element*)**, la funció **clear()** per a netejar tota la llista. A més disposa d'una funció **iterator()** o l'element **forEach**. Veurem un exemple d'això últim.

```
fun main()
{
   val persones=ArrayList<Persona>()
   persones.add(Persona("Ana",12))
   persones.add(Persona("Pedro", 15))
   persones.add(Persona("Rosa",13))
   persones.add(Persona("Andrea",15))
   val itr = persones.iterator ()
   while (itr.hasNext ()) itr.next().imprimir()
   persones.forEach{it.imprimir()}
}
```

A més per a les col·leccions podem usar un filtre **filter** que ens permetrà realitzar una selecció d'alguns elements de la col·lecció que complisquen la condició del filtre i després amb forEach es recorre el resultat. Per a l'anterior exemple, podríem fer un filtre per l'edat de 15 anys de la següent manera:

```
persones.filter{it.edad==15}.forEach{it.imprimir()}
```

Mapes

També tindrem mapes MutableMap i mapes immutables Map en kotlin. El funcionament és similar, per la qual cosa veurem els exemples dels mapes mutables.

```
fun main()
{
   val persones = mutableMapOf<String, Persona>()

   persones["21456874L"] = Persona("Ana",12)
   persones["13232345K"]=Persona("Luis", 13)
   persones.forEach{print(it.key + " ")
                    it.value.imprimir()}

}
```

En el següent exemple s'introdueixen una sèrie d'elements més. La classe **Pair** que ens permet crear parells de valors i que ajuda a afegir elements a una mapa, entre altres coses. A més es pot veure la utilització de l'operador `+=` com una altra manera d'agregar elements. I un altre exemple de filtre amb comptatge usant els elements **filter i count**.

```
fun main()
{
   val persones = mutableMapOf<String, Persona>(Pair("21456874L", Persona("Ana",12)),
                                                 Pair("13232345K",Persona("Luis", 13)))
   persones+=Pair("24568947L", Persona("Pedro",15))
   persones+=Pair("25412321L", Persona("Laura",12))
   //Filtrem per noms que comencen per L i comptem
   println(persones.filter{ it.value.nombre.startsWith('L') }.size)
   // El mateix que abans però usant count
   println(persones.count { it.value.nombre.startsWith('L') })
}
```

També es pot recórrer un mapa usant tuples i for de la següent manera:

```
 for((dni,valor) in persones)
   {
       print(dni)
       valor.imprimir()
   }
```

Hi ha diversos mètodes d'utilitat per a treballar amb mapes, alguns són iguals als que ja hem vist en les anteriors col·leccions i altres són propis, per a més informació consultar en la pàgina oficial.