Programació multimèdia
i dispositius mòbils

7. Tipus Complexos de dades

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

Continguts

1 Tipus complexos de dades

1.1 Vectors i matrius

Un vector és una col·lecció de dades del mateix tipus, agrupades sota una mateixa variable i que es distingeixen mitjançant la posició (índex) que ocupen. Per la seua banda, una matriu és un vector amb vàries dimensions.

1.1.1 Vectors i matrius en Java

Per tal de definir un vector o matriu en Java fem ús de la notació `[]`, i podem fer-ho de diverses formes:

- Realitzant la declaració primer i després la reserva de memòria:
  - Declaració:

```
int v[];    ó     int [] v;
int matriu[][]   ó   int [][] matriu;
```

  - Reserva de memòria:

```
v=new int[num];
matriu=int[numX][numY];
```

- Realitzant la declaració i reservant la memòria al mateix temps:

```java
int [] v = new int [num];
int [][] matriu = new int [numX][numY];
```

- Realitzant la declaració i assignació de valor (la reserva es fa automàticament):

```java
int [] v = {1, 2, 3}
int [][] matriu={{1, 2, 3}{4, 5, 6}}
```

Algunes de les operacions que podem realitzar amb vectors són les següents:

- Còpia

```
System.arraycopy(origen, posIniOrigen, desti, posIniDesti,longitud);
```

- Comparació

```
java.util.Arrays.compare(v, v2)
```

- Recorregut amb `for`

```
for (int item:MyArray) { System.out.println(item);}
```

1.1.2 Vectors i matrius en Kotlin

Els vectors en Kotlin es representen també amb la classe `Array`, que ja incorpora els mètodes accessors sobrecarregats per tal d'accedir amb `[]`, i d'altre components, com l'atribut `size` que ens indica la longitud.

Per tal de crear un vector, podem utilitzar la funció de llibrería `arrayOf` de la següent manera:

```
val barallaTruc = arrayOf(3, 4, 5, 6, "Manilla", "Espasa", "Bastot")
```

Si ens fixem, a diferència de Java, **els tipus de dades que conté el vector no és necessari que siguen del mateix tipus**. Recordem que Kotlin infereix el tipus de dades. En aquest cas, si detecta que el vector que definim té tipus diferents, defineix aquest com un conjunt d'elements ordenats del tipus Any(recordeu que aquest és el tipus base del què descendeixen totes les classes a Kotlin). En canvi, si assignem un tipus concret a tots els elements del vector, Kotlin inferirà que són d'aquest tipus.

Per tal d'accedir a les diferents posicions del vector, podem fer-ho amb l'operador `[]`, o bé amb `get` i `set`. Veiem, alguns exemples de tot açò:

```
>>> val v=arrayOf(1, "2", 3)          // Definim un vector
>>> v[0]="element1"                   // Modificació de la posició 0 amb []
>>> v[0]                              // Accés a la posició 0 amb []
res12: kotlin.Any = element1
>>> v.set(0, "element1modificat")     // Modificació de la posició 0 amb set
>>> v.get(0)                          // Accés a la posició 0 amb get
res14: kotlin.Any = element1modificat

>>> val v=arrayOf(1, 2, 3)            // Vector d'enters
>>> v[0]="1"                         // Error, el tipus s'ha inferit com a Int
```

```
error: type mismatch: inferred type is String but Int was expected
v[0]="1"
     ^

>>> v[0]
res18: kotlin.Int = 1                // El tipus és Int, no Any
```

Per altra banda, si volem generar un vector d'una longitud determinada, sense assignar valors inicialment, podem utilitzar la funció `arrayOfNulls()`, que ens generarà un vector de la longitud i el tipus que indiquem amb elements nuls:

```
val nomVector = arrayOfNulls<Tipus>(longitud)
```

Per exemple, per generar un vector de 5 elements de qualsevol tipus (Any), podem fer:

```
>>> val arr = arrayOfNulls<Any>(5)
```

Per altra banda, si volem obtenir una representació del contingut del vector com a cadena de caràcters, podem utilitzar el mètode `contentToStrint()`:

```
>>> barallaTruc.contentToString()
res30: kotlin.String = [3, 4, 5, 6, Manilla, Espasa, Bastot]
```

La bliblioteca estàndard de Kotlin ens ofereix també funcions per tal de crear vectors de tipus primitius: `intArrayOf()`, `charArrayOf()`, `longArrayOf()`, etc. que ens retornen una instància de les classes `IntArray`, `CharArray`, `LongArray`... Amb aquestes classes aconseguim optimitzar el rendiment, ja que evitem costos associats a les operacions de *boxing* i *unboxing* (conversió de tipus primitius a classes i viceversa).

Recorrecut de vectors

La classe *Array* en Kotlin ens ofereix diverses funcions per accedir als elements i als índex d'un vector que ens faciliten el seu recorregut.

La forma més senzilla de recórrer un vector és fent ús de l'operador `in` directament sobre el vector, per tal d'accedir al seu contingut:

```
>>> val v=arrayOf("1", 2, "hola", false)
>>> for (i in v) println (i)
1
2
hola
false
```

També podem accedir als seus índex amb la propietat `.indices` que ens retorna un rang amb el índex del vector:

```
>>> for (i in v.indices) println (i)
0
1
2
3
```

Pel que podríem accedir al contingut també amb:

```
>>> for (i in v.indices) println (v[i])
1
2
hola
false
```

I si volem accedir al mateix temps als índex i al valor, fem ús del mètode `withIndex()`:

```
>>> for ((index, valor) in v.withIndex()) println ("$index - $valor")
0 - 1
1 - 2
2 - hola
3 - false
```

1.2 Col·leccions

Les col·leccions, tant en Java com en Kotlin ens permeten gestionar conjunts d'objectes. Anem a fer un repàs sobre el framework de col·leccions en Java, i una introducció a les col·leccions en Kotlin.

1.2.1 Col·leccions en Java

El framework de col·leccions en Java està composat per diverses interfícies, classes abstractes i classes, que ens ajuden a treballar de diferent forma amb conjunts d'objectes.

Al següent diagrama podem vore la relació entre totes aquestes classes i interfícies que componen el framework:

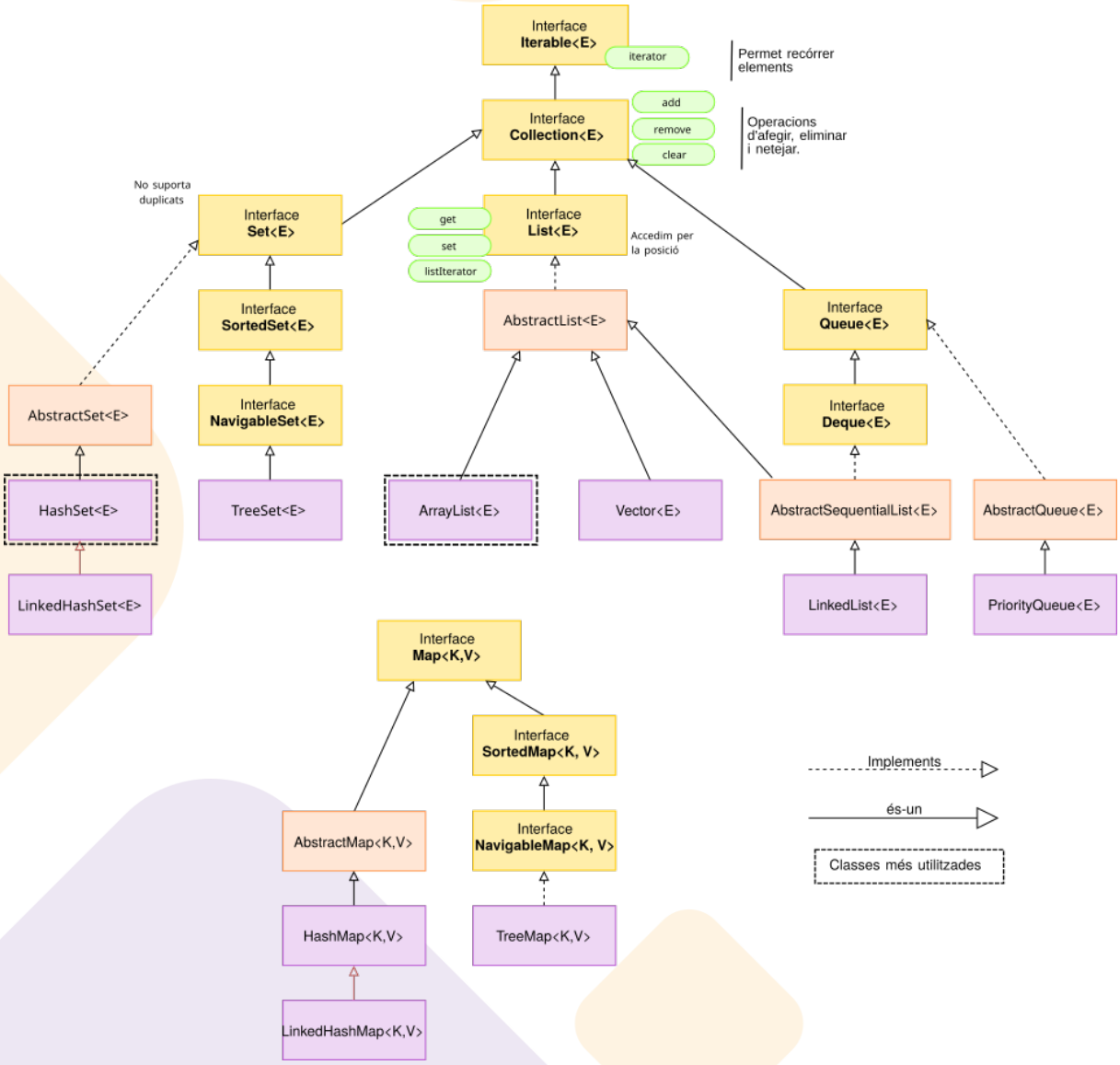

Figura 1: Jerarquia del framework de col·leccions en Java

Com veiem, tenim dues interfícies principals: *Iterable* i *Map*. La primera ens permet recórrer els elemens mitjançant un *iterador*, mentre que als *Maps* ho fem mitjançant claus.

Centrant-nos en la interfície *Iterable*, d'aquesta es deriva la interfície *Collecion*, que proporciona els mètodes addper afegir elements, `remove` per eliminar-ne, i `clear` per eliminar tots els elements. D'aquesta interfície *Collection*, es deriven tres interfícies més, però ens centrarem en dos: *Set* i *List*:

- La interfície *Set* (conjunt) ens proporciona un conjunt d'elements sense duplicats, i els seus elements en principi no tindran cap ordre. Si volem un conjunt ordenat, tenim la interfície *SortedSet* que deriva d'aquesta. El que més ens interessa d'aci és la seua classe filla abstracta *AbstractSet*, i la classe ***HashSet**, que implementa un **conjunt d'elements no ordenats i sense duplicats basant-se en hash**, la qual cosa agilitza els accessos.
- La interfície *List*, que ens proporciona una llista d'elements ordenats i accessibles a través de la seua posició. D'aquesta interfície es deriva la classe abstracta *AbstractList*, i d'aquesta, la classe ***ArrayList** que serà altra de les més utilitzades també.

Com veiem, la jerarquia de classes i interfícies per a col·leccions és bastant extensa, però ens centrarem en dues d'aquestes classes pel seu interès: *HashSet* i *ArrayList*. Anem a veure un parell d'exemples d'ús comú per recordar com utilitzaríem llistes i conjunts en Java.

Exemple amb ArrayList

- Definim un objecte de tipus List (classe abstracta) i el creem com a ArrayList que és una implementació d'aquesta.

```
List<String> noms = new ArrayList<>();
```

- Per tal de poblar la llista, podem utilitzar el mètode add

```
noms.add("Pep")
noms.add("Joan")
noms.add("Anna")
noms.add("Maria")
noms.add("Anna")    // Podem repetir elements
noms.delete("Anna") // Elimina totes les ocurrències
noms.add("Anna")
```

- Alternativament, per inicialitzar la llista a partir d'un vector, podem utilitzar el mètode Arrays.AsList, que ens torna una llista a partir d'un vector, i ens simplifica afegir elements un a un.

```
noms = Arrays.asList(new String[] {"Pep", "Joan", "Maria", "Anna"});
```

- Anem a recórrer aquest `ArrayList` fent ús de l'iterador:

- Inicialitzem un iterator que recorrerà una col·lecció d'strings a partir de la llista noms.
- La condifió del bucle serà "mentre queden elements", cosa que aconseguim amb el mètode `.hasnext()`.
- No posem cap increment al bucle, sinò que utilitzem dins el bucle el mètode `next()` per obtenir el pròxim element de la col·lecció i mostrar-lo.

```
for (Iterator<String> iterator = noms.iterator(); iterator.hasNext(); ) {
    String next = iterator.next();
    System.out.println(next);
}
```

Una altra manera de realitzar l'exemple anterior més senzilla, sería fer ús de l'operador `List.of` de Java 9, i fer ús de `ForEach`:

- Definim amb `List.of` una llista d'elements de tipus String:

```
List<String> noms=List.of("Pep", "Joan", "Maria", "Anna"");
```

- I la recorrem amb forEach, fent ús de funcions Lambda.

```
noms.forEach((p)-> {
        System.out.println(p);
    });
```

- Aquesta última funció es podría haver abreviat com a:

```
noms.forEach(System.out::println);
```

Exemple amb HashSet

- Definim objecte de tipus Set (classe abstracta) i el creem com a HashSet que és una implementació d'aquesta.

```
Set<String> conjuntNoms = new HashSet<String>();
```

- Podem afegir i eliminar elements amb els mètodes que proporciona la interfície `Collection`:

```
conjuntNoms.add("Pep");
conjuntNoms.add("Joan");
conjuntNoms.add("Maria");
conjuntNoms.add("Anna");
conjuntNoms.add("Anna"); // Tornaria false, ja que no es poden repetir
 ↪  elements
```

- A més, també podríem haver fet la inicialització amb `set.Of`:

```
conjuntNoms=Set.of("Pep", "Joan", "Maria", "Anna")
```

- Per tal de recórrer-lo, podem optar per un *for millorat*:

```
for (String nom : conjuntNoms) {
       System.out.println(nom);
}
```

- O bé a partir de Java 8, podem utilitzar `forEach`:

```
conjuntNoms.forEach(System.out::println);
```

1.2.2 Col·leccions en Kotlin

L'API de col·leccions de Kotlin és construeix sobre l'API de col·leccions de Java, tal com els ArrayLists, Maps, HasSet, etc. pel que conéixer les col·leccions de Java ens serà de gran utilitat. De tota manera, amb les col·leccions en Kotlin podrem fer més coses amb menys codi.

Anem a donar una ullada a les col·lecions més comunes: `List`, `Set` i `Map`.

De la mateixa manera que les variables, les col·leccions en Kotlin poden ser *mutables* (podem modificar-la) o impputable (no es pot modificar).

La jerarquía de classes i interfícies de les col·leccions en Kotlin està basada en la de Java:

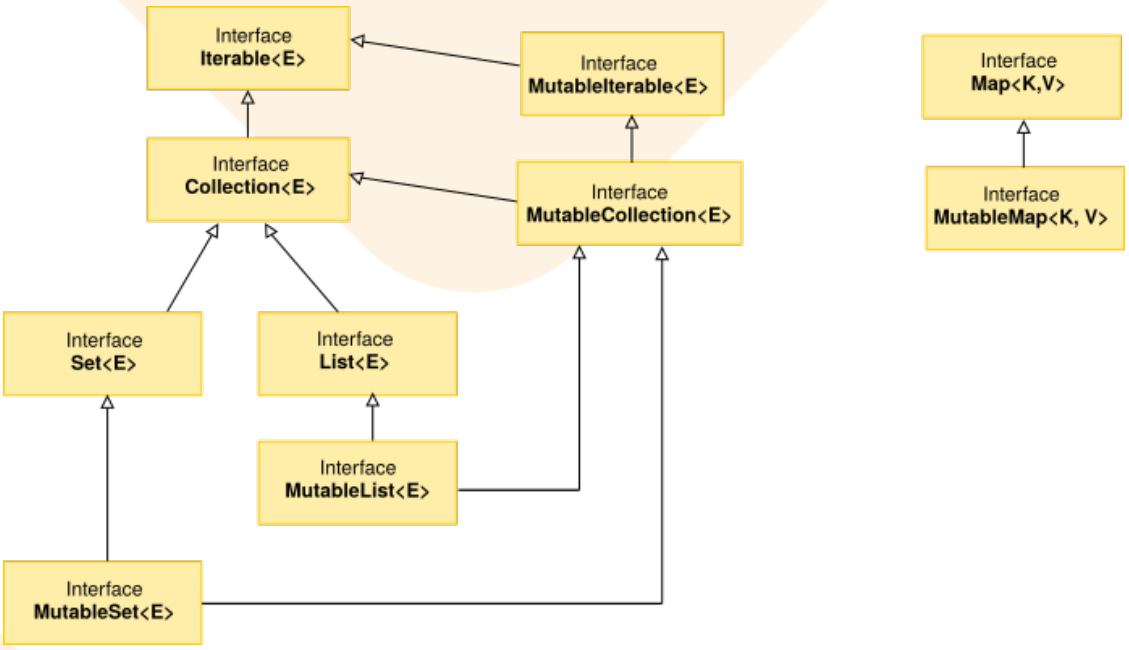

Figura 2: Col·leccions en Kotlin

- La **interfície `Iterable`** està sobre tota la jerarquia de col·leccions, i permet que els elements puguen ser representats com una seqüència d'elements i per tant aquesta puga ser recorreguda amb un `Iterator`.
- La **interfície `Collection`** extén la interfície `Iterable`, i és **immutable**. Aquesta interfície ens ofereix, entre d'altres les següent funcions i propietats:
  - `size`: Amb la longitud de la col·lecció,
  - `isEmpty()`: Tornant cert si la col·lecció no conté elements,
  - `contains(element:E)`: Ens torna cert si l'element especificat (de tipus E) es troba a la col·lecció.
  - `containsAll(element: Collection<E>)`: Ens torna cert si la la col·lecció conte **tots** els elements de la col·lecció especificada com a argument.
- Les **interfícies `Set` i `List`** extenen la interfície `Collection`.
- La **interfície `MutableIterable`** extén directament també d'`Iterable`, i ofereix un iterador mutable especialitzat d'aquesta interfície pare.
- La **interfície `MutableCollection`** habilita les col·leccions per tal que puguen ser mutables, i per tant, puguem modificar, afegir o eliminar valors. Aquesta interfície estén tant de la interfície `Collection` com de la interfície `MutableIterable`. Aquesta interfície ofereix els mètodes:
  - `add(element: E)`: Afig l'element i torna cert si s'ha afegit correctament o fals si no s'ha

afegit (per exemple si no s'admeten duplicats i ja existeix l'element),
  - `remove(element: E)`: Elimina l'element que passem com a argument. Torna cert si s'elimina i fals si no estava a la col·lecció.
  - `addAll(elements: Collection<E>)`: Afig tots els elements elements de la col·lecció que passem com a argument a la col·lecció en qüestió. Tornarà fals si no s'ha afegit cap element.
  - `removeAll(elements: Collection<E>)`: Elimina tots els elements de la col·lecció que estan a la col·lecció que passem com a argument. Tornarà fals si no s'elimina res.
  - `retainAll(elements: Collection<E>)`: Reté només els elements presents en les col·leccions, eliminant la resta d'elements si aquests existien. Tornarà fals si no reté res.
  - `clear()`: Elimina tots els elements de la col·lecció.

- Les **interfícies `MutableSet` i `MutableList`** extenen la interfície `MutableCollection`.

1.2.2.1 Llistes en Kotlin

Per crear una llista en Kotlin podem fer ús de la funció `listOf()`, que torna una **llista immutable** d'un tipus que implemente la interfície `List`.

```
var noms: List<String> = listOf("Pep", "Joan", "Anna", "Maria")
```

Per recórrer la llista i imprimir els elements fariem:

```
for (name in names) {
    println(name)
}
```

Amb la funció `listOf` també podem generar una llista d'elements de diversos tipus, sense especificar el tipus base de la llista:

```
val llistaMixta = listOf("cadena1", 'a', 1)
```

Recordeu que aquestes llistes són immutables, i per tant no es poden modificar.

Algunes altres funcions que generen llistes immutables són:

- `emptyList()`: Crea una llista immutable buïda.

- `listOfNotNull()`: Crea una llista immutable amb només els elements no nuls. És a dir, elimina els nuls de la llista que li passem (Ex.: `val nonNullsList: List<String> = listOfNotNull(2, 45, 2, null, 5, null)`)

Per altra banda, la interfície també ofereix els mètodes:

- `Lista.contains(element:E)`: Torna cert si la llista conté l'element indicat.
- `Llista.get(index: Int)`: Torna l'element en l'índex especificat,
- `Llista.indexOf(element:E)`: Retorna l'índex de la primer aparició de l'element a la llista, o -1 si no es troba.
- `Llista.lastIndexOf(element:E)`: Retorna l'índex de la última aparició de l'element en la llista, o -1 si no es troba.
- `Llista.listIterator()`: Retorna un `Iterator` sobre els elements de la llista.
- `Llista.sublist(fromIndex: Int, toIndex: Int)`: Retorna una llista que conté una part de la llista original, compresa entre els índex indicats.

Veiem alguns exemples:

```
// noms =  [Pep, Joan, Anna, Maria]
noms.get(1) // Joan
noms.indexOf("Pep") //  0
noms.size // 4
noms.contains("Anna") // true
noms.subList(1,3) // [Joan, Anna]
```

Una altra funció interessant en les llistes immutables és la que ens permet obtenir una llista mutable a partir d'una llista immutable: `toMutableList`.

```
val nomsMutable = noms.toMutableList()
```

1.2.2.2 Llistes mutables Per tal de crear llistes ***mutables***, és a dir, que es puguen modificar, eliminar i afegir elements, podem fer ús de les funció `arrayListOf()`, que ens **retornara un tipus Java ArrayList** i la funció `mutableListOf()`, que ens retornarà un tipus d'interfície *MutableList*. Aquest tipus, recordem que és una extensió de les interfícies *MutableCollection* i *List*.

```
// arrayListOf
val noms: ArrayList<String> = arrayListOf<String>("Pep", "Joan", "Anna",
 ↪  "Maria")
```

```kotlin
 // arrayListOf sense especificar-ne el tipus en la funció
val noms: ArrayList<String> = arrayListOf("Pep", "Joan", "Anna", "Maria")

//  arrayListOf inferint un tipus mixte
val items = arrayListOf("Pep", "Joan", "Anna", "Maria", 1, 2, 3)

// mutableListOf
val noms: MutableList<String> = mutableListOf<String>("Pep", "Joan", "Anna",
 ↪  "Maria")

// mutableListOf sense especificar-ne el tipus en la funció
val noms: MutableList<String> = mutableListOf<String>("Pep", "Joan", "Anna",
 ↪  "Maria")

//  mutableListOf inferint un tipus mixte
val noms = mutableListOf(1, 2, 3, "Pep", "Joan", "Anna", "Maria")
```

A efectes pràctics, tant `arrayListOf` com `mutableListOf` són equivalents. Ambdues ens creen una llista mutable i implementada per un `ArrayList` de Java. La diferència entre elles és més la intecionalitat. Així com en `arrayListOf` demanem explícitament volem una llista implementada com un `ArrayList`, en `mutableListOf`, el que demanem és una llista mutable, sense importar-nos amb què s'implemente. Actualment, Kotlin fa ús del tipus `ArrayList` per tal d'implementar aquestes llistes mutables, però en futures versions, podria modificar aquesta implementació.

Les llistes mutables (interfície *mutableList*) ens ofereixen els següents mètodes:

- Per afegir elements: `llistaMuable.add(element)`
- Per eliminar l'element en certa posició: `llistaMutable.removeAt(index)` (la primera posició és la 0)
- Per eliminar un element amb determinat valor: `llistaMutable.remove(valor)`
- Per reemplaçar el valor d'un element, farem ús de la notació de vectors: `llistaMutable[index]=valor`

1.2.2.3 Sets o conjunts Un conjunt és una col·lecció d'elements sense cap ordre establert entre ells i sense duplicats. Kotlin ofereix diverses maneres de crear conjunts, cadascuna emmagatzemada en un tipus d'estructura de dades diferent, optimitzades per a segons quines tasques.

- **La funció `setOf()`**

Crea un conjunt **immutable** que retorna una interfície Kotlin de tipus `Set`.

```
// Conjunt de tipus mixtes
val conjunt = setOf(1, 2, 3, "Pep", "Paco", "Amma")

// Conjunt d'enters
val cojuntEnters:Set<Int> = setOf(10, 20, 30)
```

- **La funció `hashSetOf()`**

Crea un conjunt **mutable** implementat com un *HashSet* de Java, que emmagatzema els elements en una taula hash, i ens permet afegir, eliminar o aïllar elements en el conjunt.

```
val conjuntEnters: java.util.HashSet<Int> = hashSetOf(1, 2, 3, 4)
conjuntEnters.add(5) //  [1, 2, 3, 4, 5]
conjuntEnters.remove(2) //  [1, 3, 4, 5]
```

- **La funció `sortedSetOf()`**

Crea un conjunt **mutable** implementat amb un *TreeSet* de Java, que ordena els elements en funció de la seua ordenació natural o per un comparador.

```
val conjuntEnters: java.util.TreeSet<Int>  = sortedSetOf(1, 2, 3, 4)
intsSortedSet.add(5) //  [1, 2, 3, 4, 5]
intsSortedSet.remove(2) //  [1, 3, 4, 5]
intsSortedSet.clear() // []
```

- **La funció `linkedSetOf()`**

Crea un conjunt **mutable** implementat amb un *LinkedHashSet* de Java, que manté una llista enllaçada d'entrades en el conjunt en l'ordre en què van ser insertades.

```
val conjuntEnters: java.util.LinkedHashSet<Int> = linkedSetOf(1, 2, 3, 4)
```

- **La funció `mutableSetOf()`**

Crea un conjunt **mutable**, i torna una interfície de tipus `MutableSet`, implementada amb un *LinkedHashSet* de Java.

```
val ConjuntEnters: MutableSet<Int> = mutableSetOf(1, 2, 3, 4)
```

1.2.2.4 Maps Els mapes són conjunts de parells de la forma *clau:valor*. Les claus han de ser úniques (per tant no hi ha duplicats), però els valors associats poden repetir-se. La implementació es realitza a través d'una col·lecció `Map` de Java.

- **La funció `mapOf()`**

Crea una col·lecció de tipus *Map* **immutable**. Per crear-la li donem una llista de parells clau:valor, i ens retornarà una interfície de tipus Kotlin *Map*. Per indicar els parells farem ús de la funció infixa `to`:

Per fer la declaració:

```
val codisPostals: Map<Int, String> = mapOf(46760 to "Tavernes de la
 ↪  Valldigna", 46410 to "Sueca", 46400 to "Cullera")
```

Per recórrer aquest mapa, farem:

```
for ((clau, valor) in codisPostals) {
    println("El codi postal de $valor és $clau")
}
```

I també podem accedir als diferents valors fent ús de la notació `[]` i utilitzant la clau:

```
println(codisPostals[46760]) // Tavernes de la Valldigna
```

Cal tindre en compte que els mapes **no es corresponen a la jerarquia de *Collection***, de fet, la interfície *Map* no estén res.

Les principals propietats i funcions d'aquesta interfície són:

- `size` que ens indica el nombre de parells que té,
- `isEmpty()` que ens diu si el mapa està buït,
- `containsKey(clau)`: Torna cert si la clau existeix al mapa,
- `containsValue(valor)`: Ens torna cert si el alguna clau del mapa conté el valor indicat com a argument.
- `get(clau)`: Ens retorna el valor associat a la clau indicada. Seria com u tilitzar l'operador `[]`. Si no es troba l'element retorna `null`.
- `keys`: Propietat que ens ofereix un `Set` immutable amb totes les claus del mapa.
- `values`: Propietat que ens ofereix una `Collection` immutable amb tots els valors del mapa.

La funció mutableMapOf():

Ens crea un mapa **mutable**, al que podrem afegir, modificar i eliminar elements. Ens retornarà una interfície Kotlin de tipus *MutableMap*.

```
val codisPostals: MutableMap<Int, String> = mutableMapOf(46760 to "Tavernes
 ↪  de la Valldigna", 46410 to "Sueca", 46400 to "Cullera")
```

Podem afegir o eliminar elements al mapa amb els mètodes `put` i `remove`:

```
codisPostals.put(46614, "Favara") // {46760=Tavernes de la Valldigna,
 ↪  46410=Sueca, 46400=Cullera, 46614=Favara}
codisPostals.remove(46614, "Favara") //  {46760=Tavernes de la Valldigna,
 ↪  46410=Sueca, 46400=Cullera}
codisPostals.remove(46400) //  {46760=Tavernes de la Valldigna, 46410=Sueca}
codisPostals.clear() // {}
```

I utilitzar les propietats i funcions pròpies de consulta dels mapes:

```
codisPostals.keys
codisPostals.values
codisPostals.get(clau) // o bé utilitzar la notació []
```

- **la funció HashMapOf()**

Ens proporicona un mapa **mutable** implementat amb un tipus Java *HashMap*, que utilitza una taula hash per implementar la interfície Java *Map*.

```
val codisPostals: java.util.HashMap<Int, String> = hashMapOf(46760 to
 ↪  "Tavernes de la Valldigna", 46410 to "Sueca", 46400 to "Cullera")
```

Aquest mapa admetrà totes les funcions de consulta i modificació pròpies de la interfície.

- **la funció linkedHashMap()**

Ens proporicona un mapa **mutable** implementat amb un tipus Java *LinkedHashMap*, que utilitza una llista enllaçada de les entradesal mapa en l'ordre en que han estat inserides.

```
val codisPostals: java.util.LinkedHashMap<Int, String> = linkedMapOf(46760
 ↪  to "Tavernes de la Valldigna", 46410 to "Sueca", 46400 to "Cullera")
```

Aquest mapa admetrà totes les funcions de consulta i modificació pròpies de la interfície.

- **la funció sortedMapOf()**

Ens proporicona un mapa **mutable** implementat amb un tipus Java *SortedMap*, que manté totes les entrades al mapa ordenades de manera ascendent per la clau.

```
val codisPostals: java.util.SortedMap<Int, String> = sortedMapOf(46760 to
 ↪  "Tavernes de la Valldigna", 46410 to "Sueca", 46400 to "Cullera")
//  S'emmagatzema: {46400=Cullera, 46410=Sueca, 46760=Tavernes de la
 ↪  Valldigna}
```

Aquest mapa admetrà totes les funcions de consulta i modificació pròpies de la interfície.

1.2.2.5 Funcions d'extesió sobre operadors Kotlin ens ofereix diverses funcions *operador* o funcions d'extensió, que poden ser invocades sobre les col·leccions. Donem una ullada a les més útils:

- La **funció `last()`** retorna l'últim element d'una col·lecció com una llista o un conjunt. Aquesta funció admet un predicat que restringeix l'operació a un subconjunt d'elements.
- De la mateixa manera, la **funció `first()`** ens retorna el primer element d'una col·lecció com una llista o un conjunt. Aquesta funció admet un predicat que restringeix l'operació a un subconjunt d'elements.

```
val llista: List<String> = listOf("un", "dos", "tres", "quatre", "cinc")

llista.last() // cinc

// Últim element de la col·lecció, que compleix la condició indicada
// Utiitzem l'operador `it` per fer referència a cada element en qüestió:
llista.last({it.length==3})  // dos (restringim l'operació als items de
 ↪  longitud 3)

// També es pot expressar sense els parèntesis
llista.last{it.length==3}

llista.first()  // un
llista.first({it.length==4}) //Tres
```

Per als conjunts, recordem que aquests no admeten duplicats, però que sí que mantenen l'ordre d'inseció. Veiem alguns exemples:

```
val conjunt: Set<Int> = setOf(2, 3, 1, 6, 6, 2)
```

```
conjunt.last() // 6
// El resultat és 6 no perquè siga el major, sinò perquè és
// l'últim element que s'ha afegit al conjut, ja que l'últim
// element que s'ha passat com a argument a setOf és un 2
// que ja estava al conjunt (i per tant no s'ha inserit)

conjunt.first() // 2

// Veiem incorporant alguns predicats:

conjunt.first({it>3}) // 6: Primer element afegit major que 3
conjunt.last({it<5}); // 1: Últim element afegit menor que 5
```

- La **funció `max()`** retorna l'element més gran o *nul* si no existeix.
- La **funció `min()`** ens retorna l'element més menut o *nul* si no existeix.

```
llista.max() //  un: Element de menor longitud
llista.min() // cinc: Element més llarg
conjunt.max() // 6
>>> conjunt.min() // 1
```

- La **funció `drop(n)`** ens retorna una nova llista o conjunt sense els primers *n* elements:

```
llista.drop(2) // [tres, quatre, cinc]
conjunt.drop(3) // [6]
```

- La **funció `plus(element)`** ens retorna una nova llista resultat d'afegir l'element indicat a la llista o conjunt. Si es tracta d'un conjunt i l'element ja està, no l'afegirà.
- La **funció `minus(element)`** ens retorna una nova llista resultat d'eliminar l'element indicat a la llista o conjunt. Si l'element ja no estava, tornarà la matixa llista o conjunt.

```
var llista2=llista.plus("hola") // llista2=[un, dos, tres, quatre, cinc,
 ↪  hola]
var conjunt2=conjunt.plus(6) //  conjunt2=[2, 3, 1, 6] (No s'ha afegit res)

var llista3=llista.minus("tres") // llista3= [un, dos, quatre, cinc]
var conjunt3=conjunt.minus(10) //  conjunt3=[2, 3, 1, 6] (no es modifica, el
 ↪  10 no estava al conjunt)
```

Podem trobar molta més informació sobre col·leccions en Kotlin a la documentació:

- https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/