

# Programmation Fonctionnelle (Scala)

# Plan du cours

- ▶ Map, flatmap, flatten
- ▶ Folding & reducing
- ▶ scan

# Le Mapping

- Le Mapping est une fonction **one to one**, où pour chaque élément d'entrée un élément de sortie est renvoyé.

```
class C[A]:  
  def map[B](f: (A) => B): C[B]
```

- La fonction *Map()* permet d'appliquer un traitement à chaque élément d'une structure.
  - **Exemple** de fonction *Map()* sur les listes

```
object Main extends App {
```

```
  val liste = List(2, 5, 7, 8, 9, 12, 15);  
  println(liste);
```

```
List(2, 5, 7, 8, 9, 12, 15)  
List(4, 10, 14, 16, 18, 24, 30)  
List(7, 10, 12, 13, 14, 17, 20)
```

```
  val liste1 = liste.map(_*2);  
  println(liste1);
```

```
  val liste2 = liste.map(x => x + 5 );  
  println(liste2);  
}
```

```
object Main extends App {
```

```
  val liste = List("A0", "B1", "C2", "D3", "E4", "F5");  
  println(liste);
```

```
  val liste3 = liste.map(x => x * 2);  
  println(liste3);  
}
```

```
List(A0, B1, C2, D3, E4, F5)  
List(A0A0, B1B1, C2C2, D3D3, E4E4, F5F5)
```

# Le Mapping

## ► Exemple de fonction *map()* sur les Map.

Nous avons vu que les *Map* sont un ensemble de paires (*clés*, *valeur*).

Il est possible d'appliquer la fonction *map()*, sur les valeurs uniquement ou sur toute la paire:

Pour appliquer un mapping sur les valeurs, utiliser la fonction *mapValues()*

Pour appliquer un mapping sur les clés uniquement, utiliser la fonction *map()*

```
object Main extends App {  
  
  val etudiants : Map[Int, String] = Map(1 -> "Ali", 2 -> "Nathalie", 3 -> "Marwa", 4 -> "Rémy", 5 -> "Anna" );  
  println(etudiants);  
  
  val etudiants1 = etudiants.mapValues(x => x + "***");  
  println(etudiants1);  
  
  val etudiants2 = etudiants.map{case (x, y) => (x+2, y*2)};  
  println(etudiants2);  
}  
  
Map(5 -> Anna, 1 -> Ali, 2 -> Nathalie, 3 -> Marwa, 4 -> Rémy)  
Map(5 -> Anna**, 1 -> Ali**, 2 -> Nathalie**, 3 -> Marwa**, 4 -> Rémy**)  
Map(5 -> MarwaMarwa, 6 -> RémyRémy, 7 -> AnnaAnna, 3 -> AliAli, 4 -> NathalieNathalie)
```

# Le Mapping

- ▶ **flatten**: une fonction qui permet de rassembler les elements de deux structures dans une seule structure.
- ▶ Elle prend une liste en paramètre et renvoie la concaténation de tous les éléments sous la forme d'une liste simple

```
object Main extends App {  
  val Liste = List(Map('A' -> "Bonjour", 'B' -> "Bonsoir"), Map('C' -> "Bonne matinée", 'D' -> "Bon après-midi"));  
  println("Liste "+Liste);  
  
  val Liste2 = Liste.flatten;  
  println("Liste2 "+Liste2);  
}
```

---

```
Liste List(Map(A -> Bonjour, B -> Bonsoir), Map(C -> Bonne matinée, D -> Bon après-midi))  
Liste2 List((A,Bonjour), (B,Bonsoir), (C,Bonne matinée), (D,Bon après-midi))
```

```
object Main extends App {  
  
  val Liste = List(List(1,2,3,4) , List(5,6,7,6),List(9,10));  
  println("Liste "+Liste);  
  
  val Liste2 = Liste.flatten;  
  println("Liste2 "+Liste2);  
}
```

---

```
Liste List(List(1, 2, 3, 4), List(5, 6, 7, 6), List(9, 10))  
Liste2 List(1, 2, 3, 4, 5, 6, 7, 6, 9, 10)
```

# Mapping

- ▶ **Flatmap**: Il peut être défini comme un mélange de méthode de **map()** et de méthode **flatten**.
- ▶ La sortie obtenue en exécutant la méthode **map()** suivie de la méthode **flatten** est la même que celle obtenue par **flatMap()**. Ainsi, nous pouvons dire que **flatMap** exécute d'abord la méthode **map()** , puis la méthode **flatten** pour générer le résultat souhaité.

# Mapping

```
object Main extends App {  
  
  val name = Seq("NATHALIE", "OSMANE");  
  println("name: " + name);  
  
  val res1 = name.map(_.toLowerCase());  
  println("res1: " + res1);  
  
  val res2 = res1.flatten;  
  println("res2: " + res2);  
  
}
```

```
name: List(NATHALIE, OSMANE)  
res1: List(nathalie, osmane)  
res2: List(n, a, t, h, a, l, i, e, o, s, m, a, n, e)
```

```
object Main extends App {  
  
  val name = Seq("NATHALIE", "OSMANE");  
  println("name: " + name);  
  
  val name1 = name.flatMap(_.toLowerCase());  
  println("name1: " + name1);  
  
  name: List(NATHALIE, OSMANE)  
  name1: List(n, a, t, h, a, l, i, e, o, s, m, a, n, e)
```

# Mapping

```
object Main extends App {  
  
  def f(x: Int) = List(x, x+1, x*2, x*x)  
  
  val liste = List(2, 4, 6, 9, 12, 14);  
  println("liste:  "+liste);  
  
  val res1 = liste.map(f);  
  println("res1:  "+res1);  
  
  val res2 = res1.flatten;  
  println("res2:  "+res2);  
  
}
```

```
object Main extends App {
```

```
  def f(x: Int) = List(x, x+1, x*2, x*x)  
  
  val liste = List(2, 4, 6, 9, 12, 14);  
  println("liste:  "+liste);  
  
  val res = liste.flatMap(f);  
  println("res:    "+res);  
}
```

```
liste: List(2, 4, 6, 9, 12, 14)
```

```
res: List(2, 3, 4, 4, 4, 5, 8, 16, 6, 7, 12, 36, 9, 10, 18, 81, 12, 13, 24, 144, 14, 15, 28, 196)
```

```
liste: List(2, 4, 6, 9, 12, 14)
```

```
res1: List(List(2, 3, 4, 4), List(4, 5, 8, 16), List(6, 7, 12, 36), List(9, 10, 18, 81), List(12, 13, 24, 144), List(14, 15, 28, 196))
```

```
res2: List(2, 3, 4, 4, 4, 5, 8, 16, 6, 7, 12, 36, 9, 10, 18, 81, 12, 13, 24, 144, 14, 15, 28, 196)
```



# Filtering

- ▶ Il est courant de parcourir une collection et d'en extraire une nouvelle collection avec des éléments qui correspondent à certains critères.
- ▶ La méthode *filtering()* est une opération *one to zero or one*

```
class C[A]:  
  def filter(f: A => Boolean): C[A]
```

```
object Main extends App {  
  
  val nombres = Map("un" -> 1, "deux" -> 2, "trois" -> 3, "quatre" -> 4, "cinq" -> 5, "six" -> 6, "sept" -> 7);  
  println("nombres: "+nombres);  
  
  val nbrs = nombres.filter{ case (x, y) => x.startsWith("s")}  
  println("nbrs:  "+nbrs);  
}
```

---

```
nombres: Map(quatre -> 4, cinq -> 5, trois -> 3, six -> 6, un -> 1, sept -> 7, deux -> 2)  
nbrs:  Map(six -> 6, sept -> 7)
```

# Filtering

- ▶ La plupart des collections qui prennent en charge le filtre ont un ensemble de méthodes associées pour récupérer le sous-ensemble d'une collection.
- ▶ `def drop(n: Int): C[A]` : retourne une nouvelle collection sans les n premiers

éléments.

```
val nombres = Map("un" -> 1, "deux" -> 2, "trois" -> 3, "quatre" -> 4, "cinq" -> 5, "six" -> 6, "sept" -> 7);  
println("nombres: "+nombres);
```

```
val nbrs = nombres.drop(3);  
println("nbrs: "+nbrs);
```

```
nombres: Map(quatre -> 4, cinq -> 5, trois -> 3, six -> 6, un -> 1, sept -> 7, deux -> 2)  
nbrs: Map(six -> 6, un -> 1, sept -> 7, deux -> 2)
```

# Filtering

- def *dropWhile* (p: (A) => Boolean): C[A] : Supprime le préfixe le plus long des éléments qui satisfont le prédicat p. La nouvelle collection renvoyée commence par le premier élément qui ne satisfait pas le prédicat.

```
val nombres = List(0, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
println("nombres: "+nombres);  
  
val nbrs = nombres.dropWhile(x=>{x % 2 == 0});  
println("nbrs:  "+nbrs);  
`
```

```
nombres: List(0, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
nbrs: List(3, 4, 5, 6, 7, 8, 9, 10)
```

# Filtering

- def *exists* (p: (A) => Boolean): Boolean : Retourner true si le prédicat est valable pour au moins un des éléments de cette collection, renvoyer false sinon.

```
val nombres = List(0, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
println("nombres: "+nombres);  
  
val rep = nombres.exists(x=>{x % 2 == 0});  
println("rep: "+rep);  
}
```

```
nombres: List(0, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
rep: true
```

# Filtering

- def *filterNot* (p: (A) => Boolean): C[A]: La négation du filtre; Sélectionnez tous les éléments qui ne satisfont pas au prédicat.

```
val nombres = List(0, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
println("nombres: "+nombres);
```

```
val rep = nombres.filterNot(x=>{x % 2 == 0});  
println("rep: "+rep);  
}
```

```
nombres: List(0, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
rep: List(3, 5, 7, 9)
```

# Filtering

► def *find* (p: (A) => Boolean): Option[A]

Recherchez le premier élément de la collection qui satisfait le prédicat, ou None s'il n'existe aucun élément satisfaisant le prédicat.

```
val nombres = List(0, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
println("nombres: "+nombres);
```

```
val rep = nombres.find(x=>{x % 2 == 0});  
println("rep: "+rep);
```

```
nombres: List(0, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
rep: Some(0)
```

# Filtering

► def *forall* (p: (A) => Boolean): Boolean

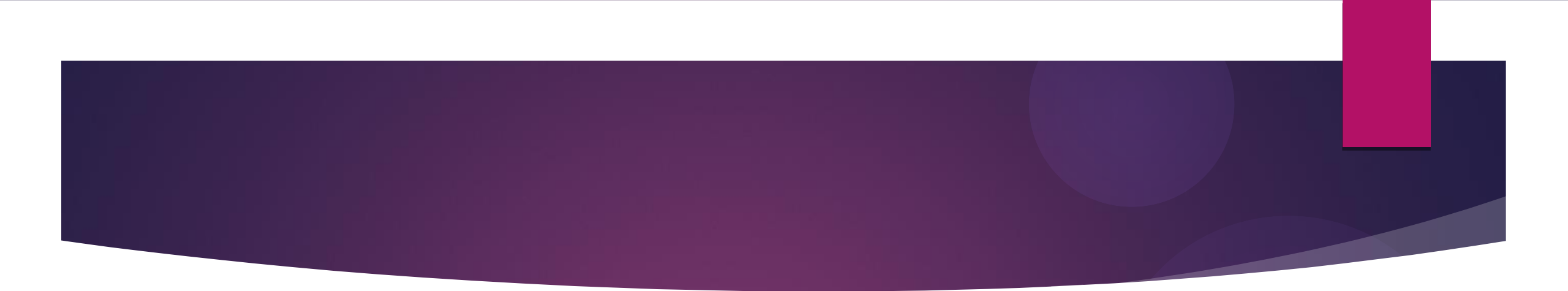
Renvoyer true si le prédicat est valable pour tous les éléments de la collection. Renvoyer false, sinon.

```
val nombres = List(0, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
println("nombres: "+nombres);  
  
val rep = nombres.forall(x=>{x % 2 == 0});  
println("rep: "+rep);  
}
```

```
nombres: List(0, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
rep: false
```

```
val nombres = List(0, 2, 4, 6, 8, 10);  
println("nombres: "+nombres);  
  
val rep = nombres.forall(x=>{x % 2 == 0});  
println("rep: "+rep);  
}
```

```
nombres: List(0, 2, 4, 6, 8, 10)  
rep: true
```



► def partition (p: (A) => Boolean): (C[A], C[A])

Partitionnez la collection en deux nouvelles collections en fonction du prédicat.

Retourner la paire de nouvelles collections où la première se compose de tous les éléments qui satisfont le prédicat et le second se compose de tous les éléments qui ne le font pas. L'ordre relatif des éléments dans les collections résultantes est le même que dans la collection originale.

```
val nombres = List(0, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
println("nombres: "+nombres);  
  
val rep = nombres.partition(x=>{x % 2 == 0});  
println("rep: "+rep);  
}
```

```
nombres: List(0, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
rep: (List(0, 2, 4, 6, 8, 10), List(3, 5, 7, 9))
```



# Filtering

► def *take* (n: Int): C[A]

Renvoyer une collection avec les n premiers éléments. Si n est supérieur à la taille de la collection, renvoyez la collection entière.

```
val nombres = List(0, 2, 3, 4, 5, 6, 7, 8, 9, 10);  
println("nombres: "+nombres);  
  
val rep = nombres.take(3);  
println("rep: "+rep);  
}
```

```
nombres: List(0, 2, 3, 4, 5, 6, 7, 8, 9, 10)  
rep: List(0, 2, 3)
```

# Filtering

► def *takeWhile* (p: (A) => Boolean): C[A]

Retourne le premier préfixe le plus long des éléments qui satisfont le prédicat.

```
val nombres = List(0, 2, 3, 4, 6, 8, 5, 9, 10);  
println("nombres: "+nombres);  
  
val rep = nombres.takeWhile(x=>{x % 2 == 0});  
println("rep: "+rep);  
}
```

```
nombres: List(0, 2, 3, 4, 6, 8, 5, 9, 10)  
rep: List(0, 2)
```

# Folding & Reducing

- ▶ Les deux sont des opérations pour réduire une collection à une collection plus petite ou à une seule valeur, de sorte qu'ils sont des opérations plusieurs-à-un.
- ▶ Le Folding commence par une valeur initiale et traite chaque élément de la collection dans le contexte de cette valeur.
- ▶ En revanche, le Reducing ne commence pas par une valeur initiale fournie par l'utilisateur. Mais il utilise l'un des éléments de la collection comme valeur initiale, généralement le premier ou le dernier élément .

# Folding & Reducing

## ► Reducing:

► *reduceLeft()*: permet de réduire la collection en commençant des valeurs gauche

```
val nombres = List(0, 2, 3, 4, 6, 8, 5, 9, 10);  
println("nombres: "+nombres);  
  
val res = nombres.reduceLeft(_ + _);  
println("res: "+res);  
  
nombres: List(0, 2, 3, 4, 6, 8, 5, 9, 10)  
res: 47
```

```
val lst = List("S", "C", "A", "L", "A");  
println("lst: "+lst);  
val lst1 = lst.reduceLeft(_+_);  
println("lst1: "+lst1);  
  
lst: List(S, C, A, L, A)  
lst1: SCALA
```

# Folding & Reducing

```
val nombres = List(0, 2, 3, 4, 6, 8, 5, 9, 10);  
println("nombres: "+nombres);  
  
val nbrs = nombres.reduceLeft((x, y) => {println(x+" ; "+y); x+y;});  
println("nbrs: "+nbrs);
```

```
nombres: List(0, 2, 3, 4, 6, 8, 5, 9, 10)  
0 ; 2  
2 ; 3  
5 ; 4  
9 ; 6  
15 ; 8  
23 ; 5  
28 ; 9  
37 ; 10  
nbrs: 47
```

```
val nombres = List(0, 2, 3, 4, 6, 8, 5, 9, 10);  
println("nombres: "+nombres);  
  
val nbrs = nombres.reduceLeft((x, y) => {println(x+"+"+y+"="+x+y); x+y;});  
println("nbrs: "+nbrs);
```

```
nombres: List(0, 2, 3, 4, 6, 8, 5, 9, 10)  
0+2=2  
2+3=5  
5+4=9  
9+6=15  
15+8=23  
23+5=28  
28+9=37  
37+10=47  
nbrs: 47
```

# Folding & Reducing

- La fonction *reduceRight()* fonctionne de la même façon que *reduceLeft()*, c'est juste le sens de réduction qui est inverse:

```
val nombres = List(0, 2, 3, 4, 6, 8, 5, 9, 10);
println("nombres: " + nombres);

val nbrs = nombres.reduceRight((x, y) => {println(x + " + " + y + " = " + (x + y)); x + y});
println("nbrs: " + nbrs);

val lst = List("S", "C", "A", "L", "A");
println("lst: " + lst);
val lst1 = lst.reduceRight(_ + _);
println("lst1: " + lst1);
```

```
nombres: List(0, 2, 3, 4, 6, 8, 5, 9, 10)
9+10=19
5+19=24
8+24=32
6+32=38
4+38=42
3+42=45
2+45=47
0+47=47
nbrs: 47
lst: List(S, C, A, L, A)
lst1: SCALA
```

# Folding & Reducing

- Dans cet exemple, les deux fonctions `reduceLeft()` et `reduceRight()` renvoient un résultat différent:

```
val nombres = List(0, 2, 3, 4, 6, 8, 5, 9, 10);  
println("nombres: "+nombres);
```

```
val nbrs = nombres.reduceRight(_-__);  
println("nbrs: "+nbrs);
```

```
nombres: List(0, 2, 3, 4, 6, 8, 5, 9, 10)  
nbrs: 1
```

```
val nombres = List(0, 2, 3, 4, 6, 8, 5, 9, 10);  
println("nombres: "+nombres);
```

```
val nbrs = nombres.reduceLeft(_-__);  
println("nbrs: "+nbrs);
```

```
nombres: List(0, 2, 3, 4, 6, 8, 5, 9, 10)  
nbrs: -47
```

# Folding & Reducing

- Folding: comme nous l'avons dit précédemment, le folding et le reducing sont Presque identiques, la seule difference reside dans le fait qu'avec le folding on peut donner une Valeur initiale contrairement au Reducing.

```
val nombres = List(0, 2, 3, 4, 6, 8, 5, 9, 10);  
println("nombres: "+nombres);  
  
val nbrs = nombres.foldRight(0)(_ - _);  
println("nbrs: "+nbrs);
```

```
nombres: List(0, 2, 3, 4, 6, 8, 5, 9, 10)  
nbrs: 1
```

```
val nombres = List(0, 2, 3, 4, 6, 8, 5, 9, 10);  
println("nombres: "+nombres);  
  
val nbrs = nombres.foldRight(10)(_ - _);  
println("nbrs: "+nbrs);
```

```
nombres: List(0, 2, 3, 4, 6, 8, 5, 9, 10)  
nbrs: -9
```



# Folding & Reducing

- La méthode scan prend une fonction d'opérateur binaire associatif comme paramètre et l'utilise pour réduire les éléments de la collection afin de créer un total cumulé. Semblable à la méthode fold, scan permet également de spécifier une valeur initiale.

```
val nombres = List(0, 2, 3, 4, 6, 8, 5, 9, 10);  
println("nombres: "+nombres);
```

```
val nbrs = nombres.scanLeft(10)(_+_);  
println("nbrs: "+nbrs);
```

```
nombres: List(0, 2, 3, 4, 6, 8, 5, 9, 10)  
nbrs: List(10, 10, 12, 15, 19, 25, 33, 38, 47, 57)
```

```
val nbrs = nombres.scanLeft(0)(_+_);  
println("nbrs: "+nbrs);
```

```
nbrs: List(0, 0, 2, 5, 9, 15, 23, 28, 37, 47)
```

# Folding & Reducing

```
val nombres = List(0, 2, 3, 4, 6, 8, 5, 9, 10);  
println("nombres: "+nombres);  
  
val nbrs = nombres.scanLeft(0)((x, y) => x + y);  
println("nbrs: "+nbrs);
```

```
nombres: List(0, 2, 3, 4, 6, 8, 5, 9, 10)  
nbrs: List(0, 0, 2, 5, 9, 15, 23, 28, 37, 47)
```

# Folding & Reducing

► Autres fonctions:

- *Product*: calcule le produit des elements d'une collection:

```
val nombres = List( 2, 3, 4, 6, 8, 5, 9, 10);    nombres: List(2, 3, 4, 6, 8, 5, 9, 10)
println("nombres: "+nombres);                  nbrs: 518400
val nbrs = nombres.product;
println("nbrs: "+nbrs);
```

- *Sum*: calcule la somme des elements d'une collection:

```
val nbrs = nombres.sum;                        nombres: List(2, 3, 4, 6, 8, 5, 9, 10)
println("nbrs: "+nbrs);                        nbrs: 47
```

# Folding & Reducing

- *mkString*: converti les éléments d'une collection en une seule chaîne de caractères:

```
val nombres = List( 2, 3, 4, 6, 8, 5, 9, 10);  
println("nombres: "+nombres);  
val nbrs = nombres.mkString;  
println("nbrs: "+nbrs);
```

```
nombres: List(2, 3, 4, 6, 8, 5, 9, 10)  
nbrs: 234685910
```

- *mkString(sep)*: converti les éléments d'une collection en une seule chaîne de caractères

```
val nbrs = nombres.mkString(" "); // séparateur donné en paramètres.  
println("nbrs: "+nbrs);
```

```
nbrs: 2 3 4 6 8 5 9 10
```

# Folding & Reducing

- *mkString(start: String, sep: String, end: String)*: Affiche tous les éléments d'une collection dans une chaîne en utilisant le début spécifié (préfixe), sep (séparateur) et end (suffixe).

```
val nombres = List( 2, 3, 4, 6, 8, 5, 9, 10);  
println("nombres: "+nombres);  
val nbrs = nombres.mkString("A", " ", "Z");  
println("nbrs: "+nbrs);
```

```
nbrs: A2 3 4 6 8 5 9 10Z
```