ntic master
revolucionamos la comunicación

UNIVERSIDAD COMPLUTENSE DE MADRID

Colecciones en Scala

Charles Flores

Colecciones en Scala

Objetivos del tema:

- Conocer las colecciones que ofrece Scala.
- Conocer las características de la colecciones: Inmutables y Mutables.
- Conocer las funciones disponibles para cada tipo de colección

1 Colecciones de Scala

Colecciones de Scala

Una colección en Scala se puede ver como un contenedor de cosas.

Como contenedor puede ser secuencial, o un conjunto lineal de elementos, o tener otros contenedores como elementos.

Las colecciones pueden ser estrictas o lazy.

Las colecciones lazy son aquellas que no consumen memoria hasta que sus elementos son accedidos.

Las colecciones de Scala se dividen entre colecciones mutables e inmutables.

Colecciones de Scala

Una colección mutable puede ser actualizada, esto significa que su estado puede cambiar: modificar, añadir o eliminar elementos de una colección.

Las colecciones inmutables, por el contrario, nunca cambian. Este tipo de colecciones tienen operaciones que simulan adiciones, eliminaciones o actualizaciones, pero esas operaciones devolverán en cada caso una colección nueva y dejarán la colección antigua sin cambios.

Las colecciones en scala se pueden encontrar en 3 paquetes y cada variante tiene sus propias características respecto a la inmutabilidad:

- `scala.collection`
- `scala.collection.mutable`
- `scala.collection.inmutable`

Colecciones de Scala

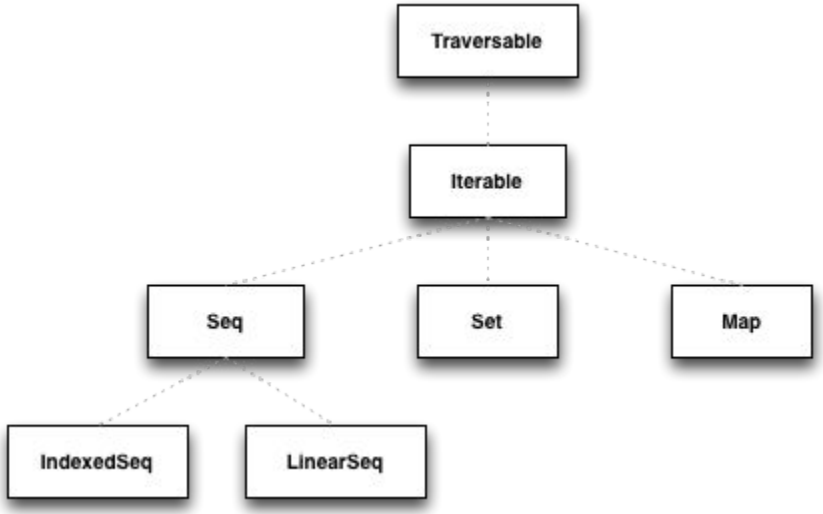
Las colecciones creadas del paquete `scala.collection.inmutable` no podrán variar una vez creadas, con lo cual se puede confiar en obtener siempre la misma colección con los mismos elementos en diferentes momentos durante la vida del programa.

Las colecciones creadas del paquete `scala.collection.mutable` cuentan con operaciones para ser modificadas. Así que tratar con colecciones mutables implica conocer qué código cambia qué colección y cuándo.

Las colecciones iterables de Scala tiene su version `inmutable` y `mutable`

Colecciones de Scala

Aquí podemos ver la jerarquía de clases de las colecciones en Scala:



2 Tuplas y Option

Tuplas

En Scala, una tupla es un valor que contiene un número fijo de elementos, cada uno puede tener un tipo distinto.

Algunas características de las tuplas:

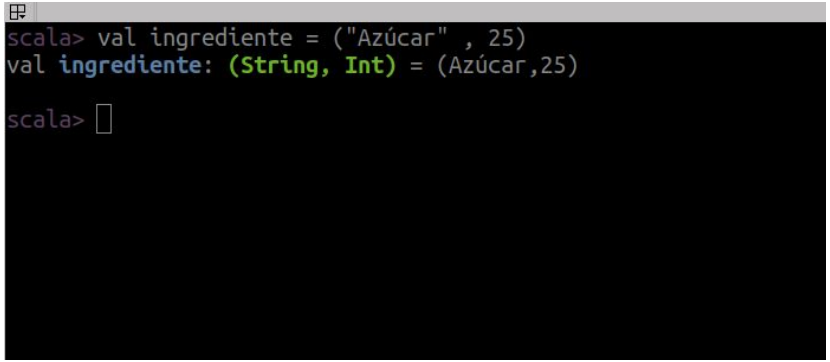
- Las tuplas son inmutables.
- Las tuplas son especialmente útiles para devolver múltiples valores de un método.
- Las tuplas son objetos de las clases de scala: Tuple2, Tuple3, Tuple4... Tuple22.
- No se puede tener una tupla con más de 22 elementos.

Tuplas

Una tupla con dos elementos puede crearse de la siguiente manera:

```
val ingrediente = ("Azúcar" , 25)
```

Creará una tupla que contendrá un elemento de tipo `String` y un elemento de tipo `Int`.



El tipo de datos de 'ingrediente' será inferido a: `Tuple2[String, Int]`, que indica que es una tupla de dos elementos cuyo primer elemento es de tipo `String` y el segundo de tipo `Int`.

Option

Option representa un valor opcional.

Es la colección que puede contener uno o cero elementos de un tipo específico.

None representa un valor faltante.

Suele ser el tipo de datos que devuelto cuando se llama a la función `.get` de las colecciones; por ejemplo:

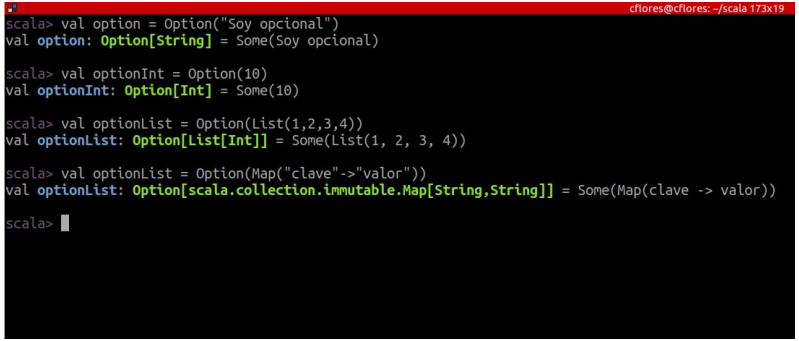
`myMap.get("key")` devolverá un Option::

- Some[T] : en caso de existir la clave "key"
- None : en caso de no encontrar la clave "key"

Option

Una variable Option[T] puede ser un objeto Some[T] o None, donde T es alguna clase definida.

El tipo Option es usado frecuentemente en programas de Scala y para hacer la comparación equivalente en java con el valor null cuando se trata de acceder a un elemento en las colecciones y no existe.



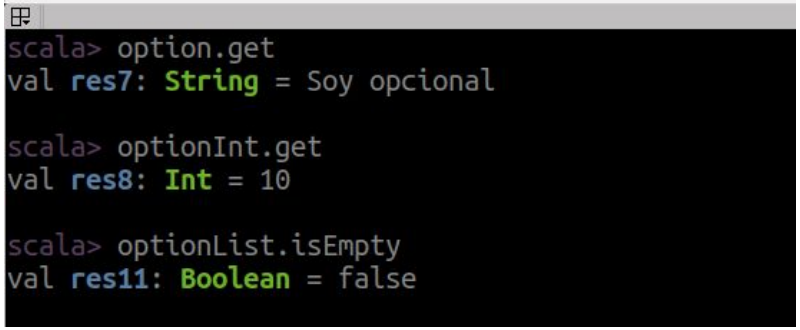
Option

Algunos de sus métodos útiles serían:

`def get: A` -> devuelve el valor de Option

`def isEmpty: Boolean` -> devuelve true si Option es None, false en cualquier otro caso.

`def exists(p: (A) => Boolean): Boolean` -> devuelve true si Option no está vacío y el la función p es evaluado a true cuando se aplica al valor de Option. Devuelve false en cualquier otro caso.

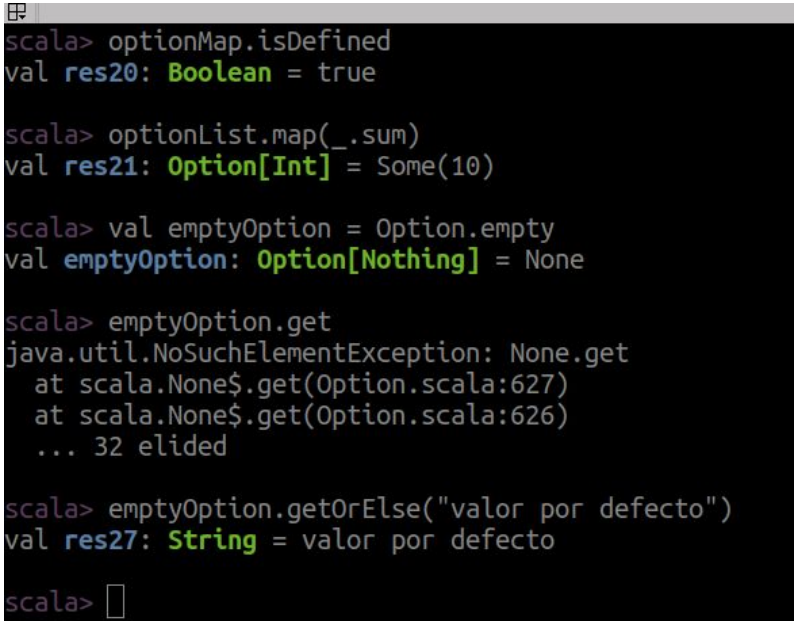


Option

`def getOrElse[B >: A](default: => B): B` -> devuelve el valor de Option en caso de existir, y si no existe devolverá el resultado de evaluar 'default'

`def isDefined: Boolean` -> devuelve true si la Option no es vacío, falso en cualquier otro caso.

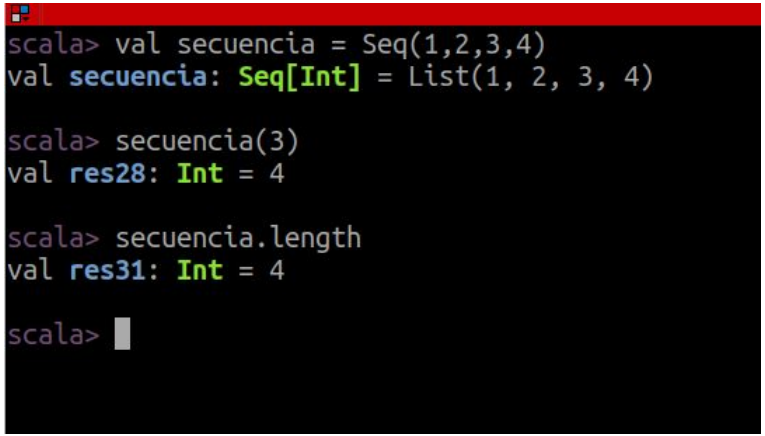
`def map[B](f: (A) => B): Option[B]` -> devuelve una instancia de Some conteniendo el resultado de la función f aplicado al valor del Option en caso de que tenga valor definido. En cualquier otro caso devuelve None.



3 Seq

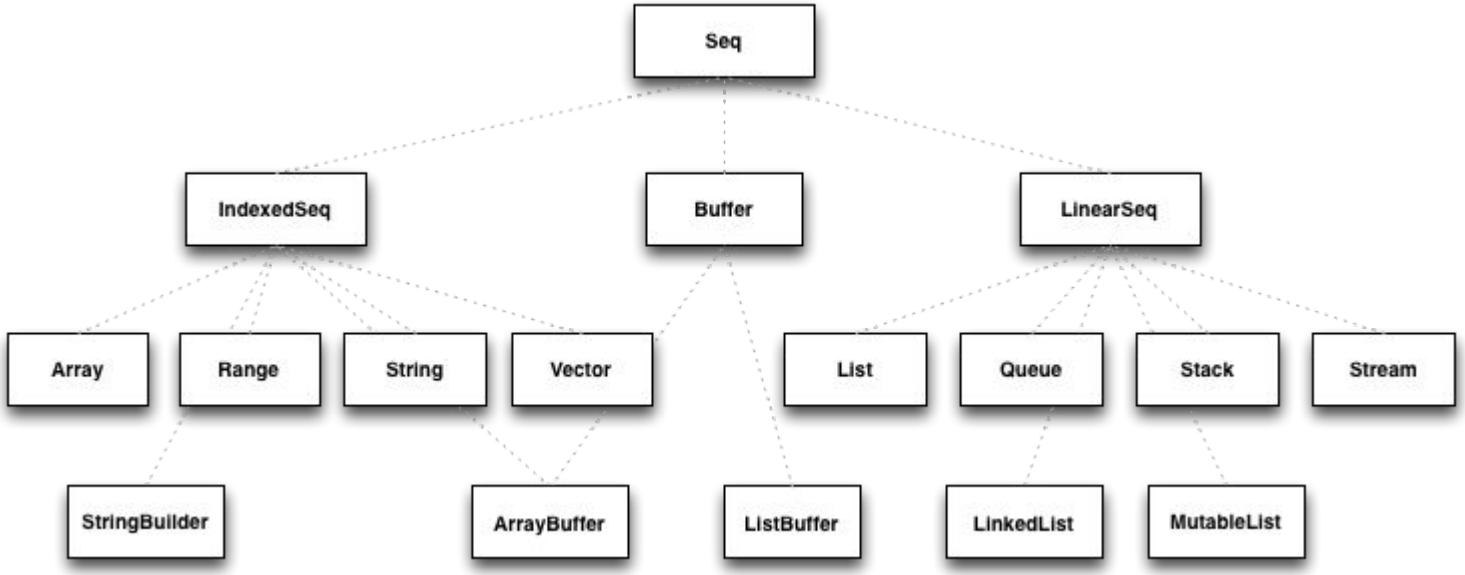
Seq

- Seq es un Trait que representa secuencias indexadas.
- Se puede acceder a sus elementos por sus índices.
- Mantiene el orden de inserción de sus elementos.
- Una secuencia es un iterable que tiene una longitud y cuyos elementos tienen posiciones de índice fijas, a partir de 0 hasta su longitud-1
- Para conocer la longitud de una secuencia se puede usar la función `length`, alias para el método `size` de las colecciones en general.



Seq

Aquí podemos ver la jerarquía de clases que heredan de Seq:



Seq

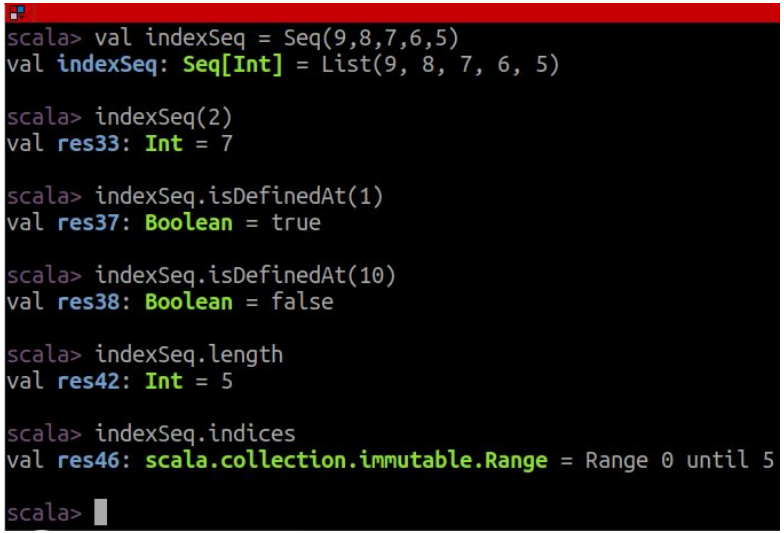
Las operaciones sobre un Seq se podrían categorizar de la siguiente manera:

- Operaciones de indexación y longitud.
- Operaciones de búsqueda de índices:
  - devuelven el índice de un elemento igual a un valor dado.
- Operaciones de adición:
  - devuelven nuevas secuencias obtenidas añadiendo elementos al principio o al final de una secuencia.
- Operaciones de actualización:
  - que devuelven una nueva secuencia obtenida mediante la sustitución de algunos elementos de la secuencia original.
- Operaciones de ordenación:
  - que clasifican los elementos de la secuencia según varios criterios.
- Comparaciones:
  - relacionan dos secuencias o buscan un elemento en una secuencia.

Seq

Operaciones de indexación y longitud:

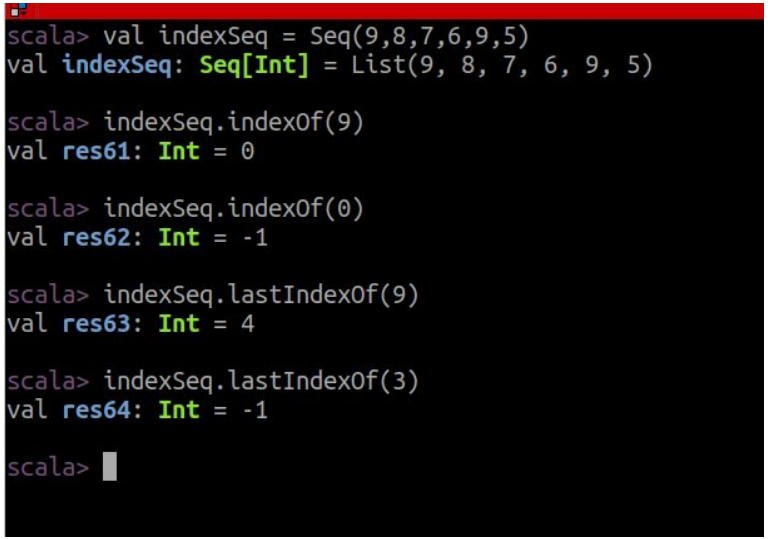
- apply:
  - de acceso directo por índice
- isDefinedAt:
  - verifica la existencia por índice
- length:
  - devuelve el tamaño de la colección
- indices:
  - devuelve una colección con los índices disponibles



Seq

Operaciones de búsqueda de índices:

- `indexOf`:
  - devuelve la primera posición encontrada para el elemento que se le pasa por argumento en caso de existir en la colección. Devuelve -1 si no encuentra un índice par el elemento.
- `lastIndexOf`:
  - similar a la anterior, pero en este caso devuelve la última posición y -1 si no lo encuentra



Existen más opciones como:

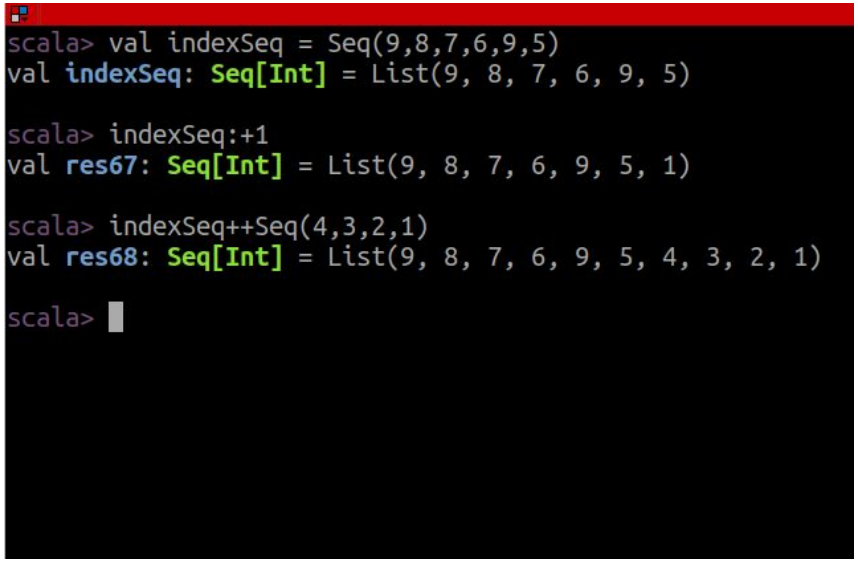
- `lastIndexOfSlice`,
- `indexWhere`,
- `lastIndexWhere`,
- `segmentLength`,
- `prefixLength`

Seq

Operaciones de adición:

- ':+'
  - crea una colección nueva donde añade un elemento nuevo a la colección
- '++'
  - genera una nueva colección de concatenar dos colecciones

Ambas operaciones no modifican la colección original, sino que devuelven una nueva colección actualizada.

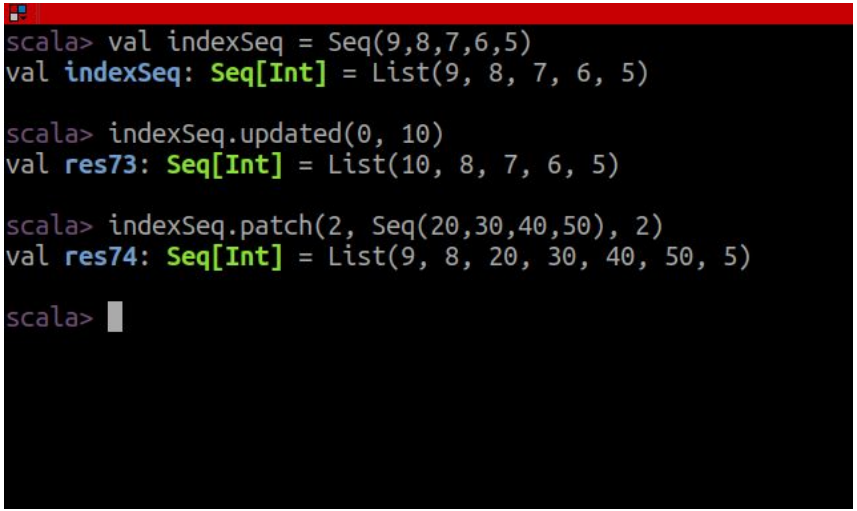


Seq

Operaciones de actualización:

- `updated`:
  - devuelve una copia de la secuenci con un solo elemento reemplazado por el valor que recibe como argumento.

- `patch`:
  - produce una nueva secuencia en la que una parte de los elementos de esta secuencia es reemplazada por otra secuencia que se le pasa por parámetro



Seq

Operaciones de ordenación:

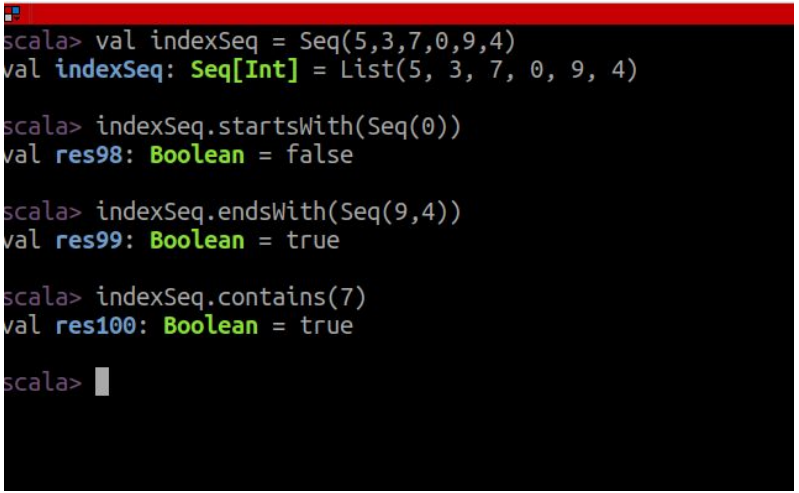
- sorted:
  - devuelve una colección nueva con los elementos de la colección original. Por defecto ordena los elementos de forma ascendente.
- sortWith:
  - acepta una función que determine el criterio de ordenación y devuelve una colección nueva con los elementos ordenados de la colección original.

```
scala> val indexSeq = Seq(5,3,6,7,2,1,0,9,4,8)
val indexSeq: Seq[Int] = List(5, 3, 6, 7, 2, 1, 0, 9, 4, 8)

scala> indexSeq.sorted
val res87: Seq[Int] = List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> indexSeq.sortWith(_<_)
val res88: Seq[Int] = List(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

scala> indexSeq.sortWith(_>_)
val res89: Seq[Int] = List(9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
```

Seq

Comparaciones: Estas funciones devuelven un Boolean en lugar de nuevas colecciones, como los casos anteriores

- startsWith:
  - comprueba si la colección original empieza por los elementos de la colección pasada por argumento.
- endsWith:
  - comprueba si la colección original acaba por los elementos de la colección pasada por argumento.
- contains:
  - comprueba si la colección original contiene un elementos con el mismo valor que el argumento que recibe.



Seq

Los ejemplos vistos anteriormente tratan con colecciones inmutables.
Si una secuencia es mutable, se ofrece además un método de actualización, `update`, que permite actualizar los elementos de la secuencia.

Nótese que **`update`** != **`updated`**.

- `update`: actualiza un elemento de la secuencia en su posición original modificando la secuencia y sólo está disponible en las secuencias mutables.

- `updated`: está disponible para todas las secuencias, pero esta función siempre devuelve una secuencia nueva con los valores actualizados respecto a la secuencia original, dejando la original sin modificar.

4 Sets

Sets

Es una colección de elementos diferentes de un mismo tipo.
Un Set es una colección que no contiene elementos duplicados.
Existen los Sets mutables e inmutables.
Por defecto, cuando se instancia un Set, éste es inmutable, en caso de querer hacerlo mutable, es necesario importar explícitamente la clase:

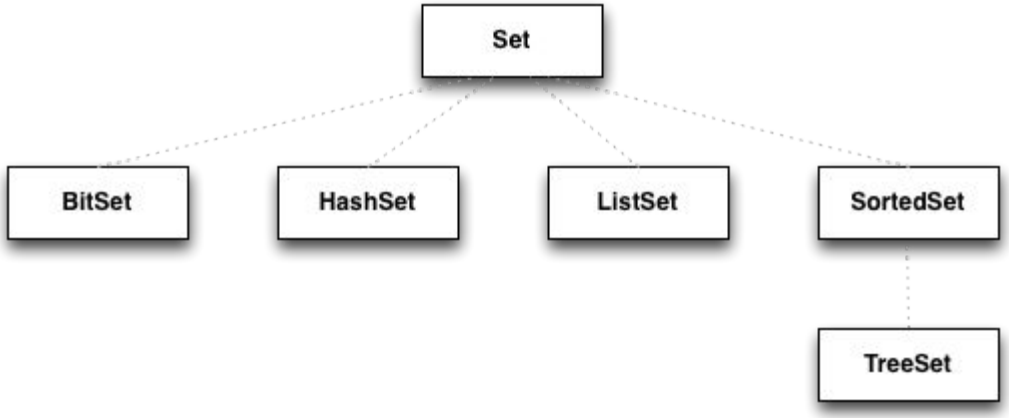
- `scala.collection.mutable.Set`

Si por necesidad se necesita instanciar ambos tipos:

```
val mutableSet = scala.collection.mutable.Set("este", "es", "un", "set", "mutable")
val inmutableSet = Set("este", "es", "un", "set", "inmutable")
```


Sets

Al igual que con Seq, de Set también heredan otras colecciones, veamos la jerarquía:



Sets

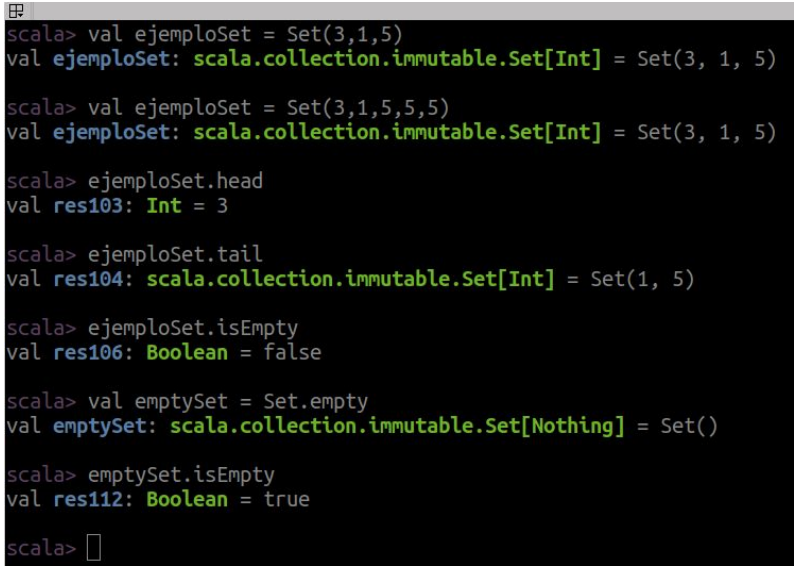
Las operaciones sobre un Seq se podrían categorizar de la siguiente manera:

- Operaciones básicas de acceso.
- Operaciones de verificación:
  - el método pregunta si un set contiene un elemento determinado.
- Operaciones de adición:
  - que añaden uno o más elementos a un conjunto, dando lugar a un nuevo conjunto.
- Operaciones de eliminación:
  - que eliminan uno o más elementos de un conjunto, dando lugar a un nuevo conjunto.
- Operaciones de conjuntos:
  - para la unión, la intersección, y la diferencia de conjuntos. Cada una de estas operaciones existe en dos formas: alfabética y simbólica.

Sets

Operaciones básicas de acceso:

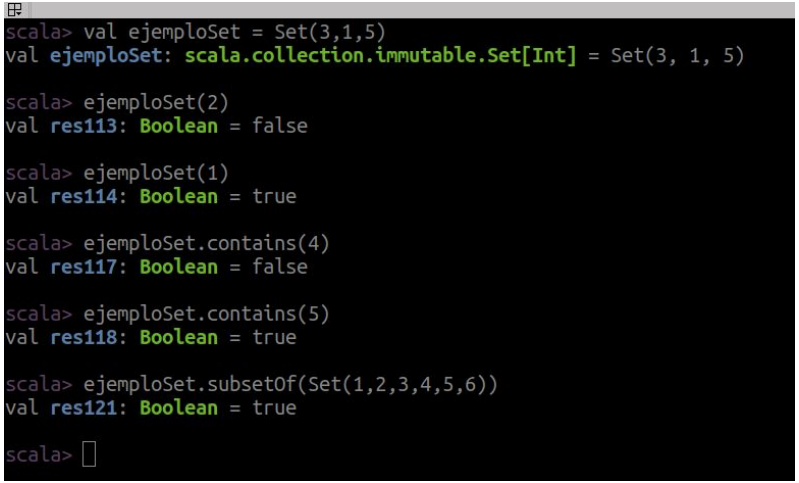
- head:
  - devuelve el primer elemento de un conjunto.
- tail:
  - devuelve una instancia de Set formado por todos los elementos excepto el primero.
- isEmpty:
  - devuelve verdadero si el conjunto está vacío, de lo contrario falso.



Sets

Operaciones de verificación:

- contains:
  - verifica si existe el elemento en el Set.
- apply = ():
  - verifica si el argumento existe como elemento en el Set
- subsetOf:
  - verifica si el set es un subconjunto de un Set pasado como argumento.



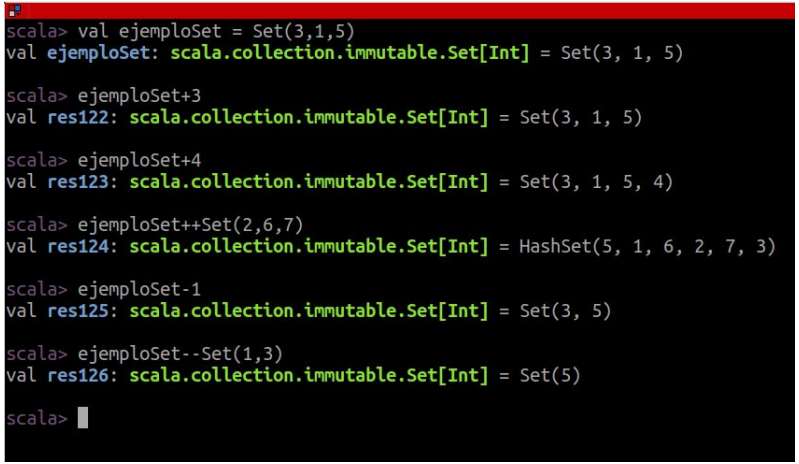
Sets

Operaciones de adición:

- +
  - añadir elementos
- ++
  - operación de concatenación

Operaciones de eliminación:

- -
  - elimina un elemento
- --
  - operación de elIminar un subconjunto de los elementos



Sets

Operaciones de conjuntos:

- intersect / &
  - devuelve la intersección de dos Sets
- union / |
  - devuelve la unión de dos Sets
- diff / &~
  - devuelve la diferencia

```
scala> val set1 = Set(1,2,3,4,5)
val set1: scala.collection.immutable.Set[Int] = HashSet(5, 1, 2, 3, 4)

scala> val set2= Set(4,5,6,7)
val set2: scala.collection.immutable.Set[Int] = Set(4, 5, 6, 7)

scala> set1 intersect set2
val res2: scala.collection.immutable.Set[Int] = HashSet(5, 4)

scala> set1 & set2
val res3: scala.collection.immutable.Set[Int] = HashSet(5, 4)

scala> set1 union set2
val res4: scala.collection.immutable.Set[Int] = HashSet(5, 1, 6, 2, 7, 3, 4)

scala> set1 |set2
val res5: scala.collection.immutable.Set[Int] = HashSet(5, 1, 6, 2, 7, 3, 4)

scala> set1 diff set2
val res6: scala.collection.immutable.Set[Int] = HashSet(1, 2, 3)

scala> set2 diff set1
val res7: scala.collection.immutable.Set[Int] = Set(6, 7)

scala>
```


Sets

Los Sets mutables también ofrecen las operaciones + y ++ para la adición de elementos y las operaciones - y -- para la eliminación de elementos, aunque se usan menos por rendimiento, ya que implican copiar el set.
Una alternativa más eficiente para añadir o eliminar elementos en los mutable Sets son: +=, -=, add y remove.

- `s += elem`: añade elem al conjunto s, y devuelve el conjunto mutado como resultado.

- `s -= elem`: elimina elem del conjunto, y devuelve el conjunto mutado como resultado.

- `s.add(elem)`: añade elem al conjunto s, y devuelve true si la operación tuvo efecto en s. Devuelve false en otro caso.

- `s.remove(elem)`: elimina elem del conjunto s, y devuelve true si la operación tuvo efecto en s. Devuelve false en otro caso.

```
scala> val setMutable = scala.collection.mutable.Set(1,2,3)
val setMutable: scala.collection.mutable.Set[Int] = HashSet(1, 2, 3)

scala> setMutable
val res25: scala.collection.mutable.Set[Int] = HashSet(1, 2, 3)

scala> setMutable += 4
val res26: setMutable.type = HashSet(1, 2, 3, 4)

scala> setMutable
val res27: scala.collection.mutable.Set[Int] = HashSet(1, 2, 3, 4)

scala> setMutable -= 1
val res28: setMutable.type = HashSet(2, 3, 4)

scala> setMutable.add(5)
val res29: Boolean = true

scala> setMutable
val res30: scala.collection.mutable.Set[Int] = HashSet(2, 3, 4, 5)

scala> setMutable.remove(3)
val res31: Boolean = true

scala> setMutable
val res32: scala.collection.mutable.Set[Int] = HashSet(2, 4, 5)

scala> setMutable.remove(6)
val res33: Boolean = false

scala> setMutable
val res34: scala.collection.mutable.Set[Int] = HashSet(2, 4, 5)
```

5 Maps

Maps

La colección `Map` de Scala, es una colección de pares clave/valor.
Cualquier valor puede ser recuperado en base a su clave.
Las claves son únicas en el mapa, pero los valores no tienen por qué ser únicos.
Los mapas también se llaman tablas Hash.
Hay dos tipos de maps, inmutable y mutable.
Por defecto, Scala usa el `Map` immutable.
Si quiere usar el Mapa mutable, se tendrá que importar explícitamente la clase:

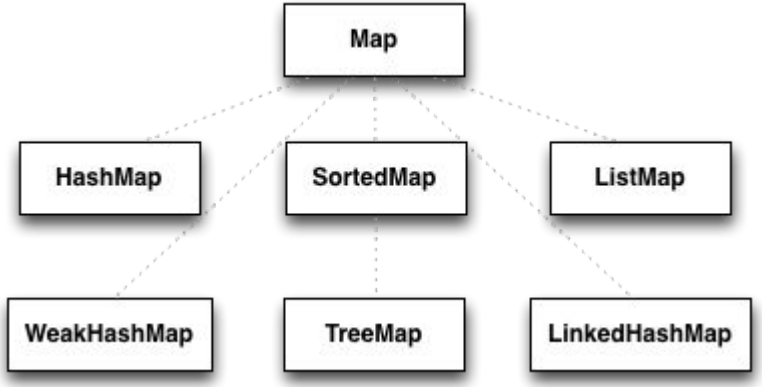
- `scala.collection.mutable.Map`

Si por necesidad se necesita instanciar ambos tipos:

```
val mutableMap = scala.collection.mutable.Map("a"->1, "b"->2, "c"->3)
val inmutableMap = Map(("a"->1), ("b"->2), ("c"->3))
```


Maps

De `Map` también existen clases que le heredan:



Maps

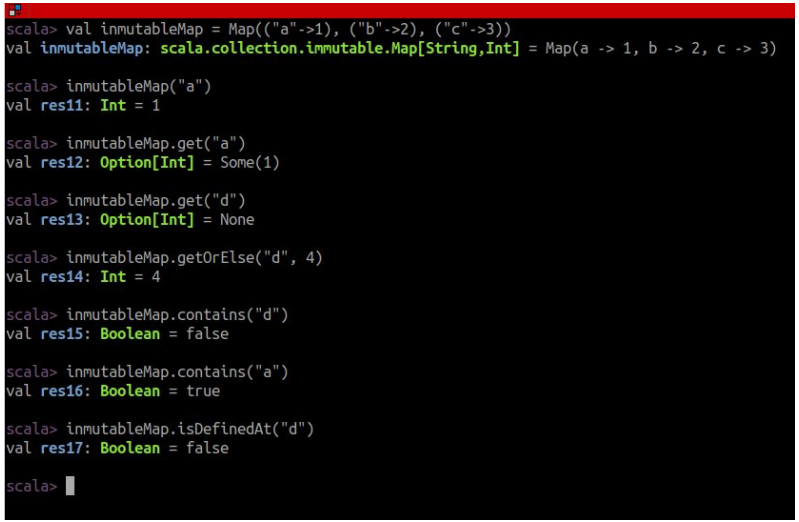
las operaciones sobre un `Map` de Scala son muy parecidas a las de los `Sets`, en realidad, como se puede observar con las siguientes categorías:

- Operaciones de búsqueda
- Adiciones y actualizaciones:
  - que te permiten añadir nuevas entradas clave-valor a un mapa o cambiar las existentes.
- Eliminación:
  - que eliminan las entradas de un mapa.
- Productores de subcolecciones:
  - que devuelven las claves y valores de un mapa por separado en varias formas.
- Transformaciones:
  - que producen un nuevo mapa filtrando y transformando las uniones de un mapa existente.

Maps

Operaciones de búsqueda:

- apply = ():
  - se puede acceder al valor de una clave directamente
- get:
  - devuelve un Option con el valor de la clave, Some[T] si existe, None en caso contrario.
- getOrElse:
  - devuelve el valor de la clave que se pida o un valor por defecto.
- contains:
  - devuelve un true en caso de que la clave exista
- isDefinedAt:
  - devuelve un true al encontrar la clave.



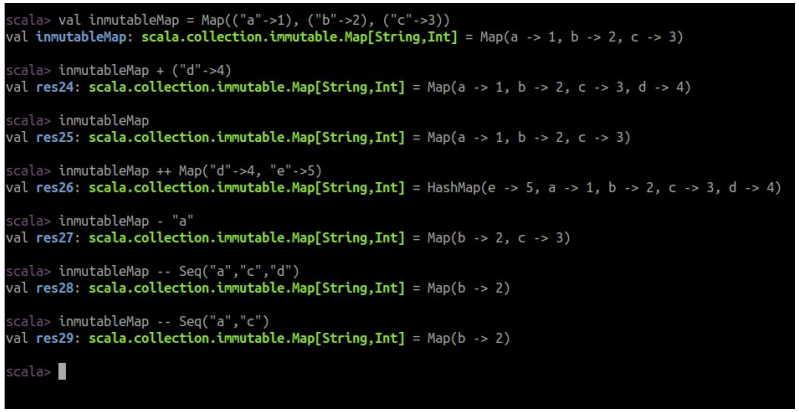
Maps

Adiciones y actualizaciones:

- +:
  - devuelve un mapa nuevo con el par clave-valor añadido al Map original.
- ++:
  - devuelve un mapa nuevo combinando dos mapas.

Eliminación:

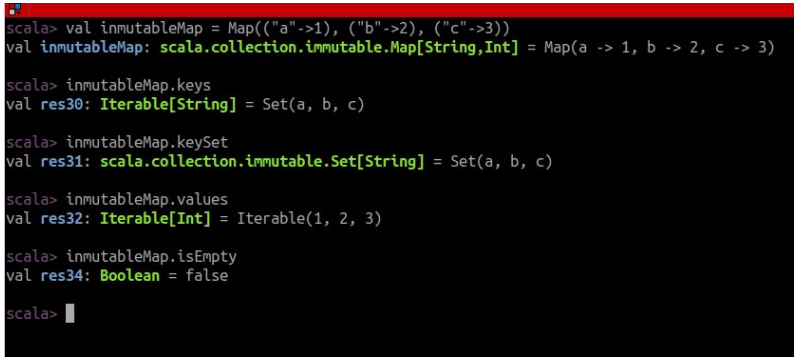
- - :
  - devuelve un mapa nuevo sin clave pasada como argumento.
- --
  - devuelve un mapa nuevo sin las claves que estén presentes en la colección pasada como argumento.



Maps

Productores de subcolecciones:

- keys:
  - este método devuelve un iterable con las claves del mapa.
- keySet
  - este método devuelve un Set con las claves del mapa.
- values
  - este método devuelve un iterable los valores del mapa.



6 Arrays y List

Arrays

Array es un tipo especial de colección en Scala.
Tienen correspondencia 1-1 con los Arrays de Java.
- Scala: `Array[Int]` == Java: `int[]`

Pero al mismo tiempo, los Array de Scala ofrecen mucho más que sus análogos de Java.

- pueden ser genéricos. Es decir, puedes tener un Array[T], donde T es un parámetro de tipo o tipo abstracto.
- los arrays de Scala son compatibles con las secuencias de Scala, puedes pasar un Array[T] donde se requiere una Seq[T].
- son compatibles con todas las operaciones de la colección Seq.

Lists

Las listas de Scala son bastante similares a los Arrays.

Las listas son inmutables, lo que significa que los elementos de una lista no pueden ser modificados por asignación.

Todos los elementos de una lista son del mismo tipo.
Las listas en Scala son listas enlazadas, a diferencia que los arrays que son listas planas.

El tipo de una lista que tiene elementos de tipo T se escribe como List[T].

ntic master
revolucionamos la comunicación

UNIVERSIDAD COMPLUTENSE DE MADRID