



ÉCOLE POLYTECHNIQUE  
FÉDÉRALE DE LAUSANNE

# Recap: Collections

Principles of Reactive Programming

Martin Odersky

## Recap: Collections

Scala has a rich hierarchy of collection classes.



## Recap: Collection Methods

All collection types share a common set of general methods.

Core methods:

`map`

`flatMap`

`filter`

and also

`foldLeft`

`foldRight`

## Idealized Implementation of map on Lists

```
abstract class List[+T] {  
  
  def map[U](f: T => U): List[U] = this match {  
    case x :: xs => f(x) :: xs.map(f)  
    case Nil => Nil  
  }  
}
```

## Idealized Implementation of flatMap on Lists

```
abstract class List[+T] {  
  
  def flatMap[U](f: T => List[U]): List[U] = this match {  
    case x :: xs => f(x) ++ xs.flatMap(f)  
    case Nil => Nil  
  }  
}
```

## Idealized Implementation of filter on Lists

```
abstract class List[+T] {  
  
  def filter(p: T => Boolean): List[T] = this match {  
    case x :: xs =>  
      if (p(x)) x :: xs.filter(p) else xs.filter(p)  
    case Nil => Nil  
  }  
}
```

## Idealized Implementation of filter on Lists

```
abstract class List[+T] {  
  
  def filter(p: T => Boolean): List[T] = this match {  
    case x :: xs =>  
      if (p(x)) x :: xs.filter(p) else xs.filter(p)  
    case Nil => Nil  
  }  
}
```

In practice, the implementation and type of these methods are different in order to

- ▶ make them apply to arbitrary collections, not just lists,
- ▶ make them tail-recursive on lists.

## For-Expressions

Simplify combinations of core methods `map`, `flatMap`, `filter`.

Instead of:

```
(1 until n) flatMap (i =>
  (1 until i) filter (j => isPrime(i + j)) map
    (j => (i, j)))
```

one can write:

```
for {
  i <- 1 until n
  j <- 1 until i
  if isPrime(i + j)
} yield (i, j)
```



## Translation of For (1)

The Scala compiler translates for-expressions in terms of `map`, `flatMap` and a lazy variant of `filter`.

Here is the translation scheme used by the compiler

1. A simple for-expression

```
for (x <- e1) yield e2
```

is translated to

```
e1.map(x => e2)
```

## Translation of For (2)

### 2. A for-expression

```
for (x <- e1 if f; s) yield e2
```

where  $f$  is a filter and  $s$  is a (potentially empty) sequence of generators and filters, is translated to

```
for (x <- e1.withFilter(x => f); s) yield e2
```

(and the translation continues with the new expression)

You can think of `withFilter` as a variant of `filter` that does not produce an intermediate list, but instead filters the following `map` or `flatMap` function application.

## Translation of For (3)

### 3. A for-expression

```
for (x <- e1; y <- e2; s) yield e3
```

where  $s$  is a (potentially empty) sequence of generators and filters, is translated into

```
e1.flatMap(x => for (y <- e2; s) yield e3)
```

(and the translation continues with the new expression)

## For-expressions and Pattern Matching

The left-hand side of a generator may also be a pattern.

### Example

```
val data: List[JSON] = ...
for {
  JObj(bindings) <- data
  JSeq(phones) = bindings("phoneNumbers")
  JObj(phone) <- phones
  JStr(digits) = phone("number")
  if digits startsWith "212"
} yield (bindings("firstName"), bindings("lastName"))
```

## Translation of Pattern Matching in For

If `pat` is a pattern with a single variable `x`, we translate

```
pat <- expr
```

to:

```
x <- expr withFilter {  
  case pat => true  
  case _ => false  
} map {  
  case pat => x  
}
```

## Exercise

```
for {  
  x <- 2 to N  
  y <- 2 to x  
  if (x % y == 0)  
} yield (x, y)
```

The expression above expands to which of the following two expressions?

☒ (2 to N) flatMap (x =>  
 (2 to x) withFilter (y =>  
 x % y == 0) map (y => (x, y)))

☐ (2 to N) map (x =>  
 (2 to x) flatMap (y =>  
 if ((x % y) == 0) (x, y)