

Other Collections

Principles of Functional Programming

Other Sequences

We have seen that lists are *linear*: Access to the first element is much faster than access to the middle or end of a list.

The Scala library also defines an alternative sequence implementation, `Vector`.

This one has more evenly balanced access patterns than `List`.

Operations on Vectors

Vectors are created analogously to lists:

```
val nums = Vector(1, 2, 3, -88)
val people = Vector("Bob", "James", "Peter")
```

They support the same operations as lists, with the exception of `::`:

Instead of `x :: xs`, there is

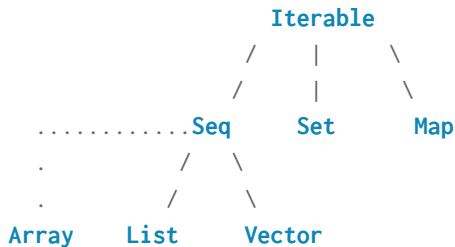
- `x +: xs` Create a new vector with leading element `x`, followed by all elements of `xs`.
- `xs :+ x` Create a new vector with trailing element `x`, preceded by all elements of `xs`.

(Note that the `:` always points to the sequence.)

Collection Hierarchy

A common base class of List and Vector is Seq, the class of all *sequences*.

Seq itself is a subclass of Iterable.



Arrays and Strings

Arrays and Strings support the same operations as Seq and can implicitly be converted to sequences where needed.

(They cannot be subclasses of Seq because they come from Java)

```
val xs: Array[Int] = Array(1, 2, 3)
```

```
xs.map(x => 2 * x)
```

```
val ys: String = "Hello world!"
```

```
ys.filter(_.isUpper)
```

Ranges

Another simple kind of sequence is the *range*.

It represents a sequence of evenly spaced integers.

Three operators:

to (inclusive), until (exclusive), by (to determine step value):

```
val r: Range = 1 until 5
```

```
val s: Range = 1 to 5
```

```
1 to 10 by 3
```

```
6 to 1 by -2
```

A Range is represented as a single object with three fields: lower bound, upper bound, step value.

Some more Sequence Operations:

<code>xs.exists(p)</code>	true if there is an element x of xs such that $p(x)$ holds, false otherwise.
<code>xs.forall(p)</code>	true if $p(x)$ holds for all elements x of xs , false otherwise.
<code>xs.zip(ys)</code>	A sequence of pairs drawn from corresponding elements of sequences xs and ys .
<code>xs.unzip</code>	Splits a sequence of pairs xs into two sequences consisting of the first, respectively second halves of all pairs.
<code>xs.flatMap(f)</code>	Applies collection-valued function f to all elements of xs and concatenates the results
<code>xs.sum</code>	The sum of all elements of this numeric collection.
<code>xs.product</code>	The product of all elements of this numeric collection
<code>xs.max</code>	The maximum of all elements of this collection (an Ordering must exist)
<code>xs.min</code>	The minimum of all elements of this collection

Example: Combinations

To list all combinations of numbers x and y where x is drawn from $1..M$ and y is drawn from $1..N$:

```
(1 to M).flatMap(x =>
```


Example: Combinations

To list all combinations of numbers x and y where x is drawn from $1..M$ and y is drawn from $1..N$:

```
(1 to M).flatMap(x => (1 to N).map(y => (x, y)))
```

Example: Scalar Product

To compute the scalar product of two vectors:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =  
  xs.zip(ys).map((x, y) => x * y).sum
```

Example: Scalar Product

To compute the scalar product of two vectors:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =  
  xs.zip(ys).map((x, y) => x * y).sum
```

Note that there is some automatic decomposition going on here.

Each pair of elements from `xs` and `ys` is split into its halves which are then passed as the `x` and `y` parameters to the lambda.

Example: Scalar Product

If we wanted to be more explicit, we could also write scalar product like this:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =  
  xs.zip(ys).map(xy => xy._1 * xy._2).sum
```

Example: Scalar Product

On the other hand, if we wanted to be more even more concise, we could also write it like this:

```
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =  
  xs.zip(ys).map(_ * _).sum
```

Exercise:

A number n is *prime* if the only divisors of n are 1 and n itself.

What is a high-level way to write a test for primality of numbers? For once, value conciseness over efficiency.

```
def isPrime(n: Int): Boolean = ???
```

Exercise:

A number n is *prime* if the only divisors of n are 1 and n itself.

What is a high-level way to write a test for primality of numbers? For once, value conciseness over efficiency.

```
def isPrime(n: Int): Boolean =  
  (2 to n - 1).forall(d => n % d != 0)
```