Updated: 2021-09-11

2.2 Let’s write some more code

Scala expressions

- First of all, as we will see in 2.3, Scala programs are also expressions.

- So let's take a look at an expression. Follow along with me in the REPL.

```
scala> 1 + 1
res0: Int = 2

scala> res0 * res0
res1: Int = 4

scala> 2 * 2
res2: Int = 4
```

Expressions (2)

```
scala> "Hello World!"
res3: String = Hello World!
```

```
scala> val hw = "Hello World!"
hw: String = Hello World!
```

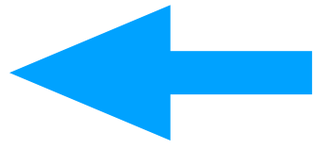
```
scala> "Hello World!\n" * 3
res4: String =
"Hello World!
Hello World!
Hello World!
"
scala> s"$res3\n" * 3
res5: String =
"Hello World!
Hello World!
Hello World!
"
```

Refactoring: extraction

```
scala> 2 * 2
res6: Int = 4
def square(x: Int) = x * x
square: (x: Int)Int
scala> square(2)
res7: Int = 4

scala> square("Hello World!")
<console>:13: error: type mismatch;
 found   : String("Hello World!")
 required: Int
       square("Hello World!")
              ^

scala> def square(x: Double) = x * x
square: (x: Double)Double
scala> square(2)
res8: Double = 4.0
```

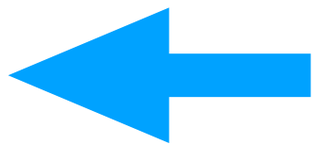
Lists

```
scala> List(1,2,3)
res9: List[Int] = List(1, 2, 3)

scala> res9 foreach println
1
2
3
```



This is *not* an expression!
We are invoking a side-effect

```
scala> def square(x: Int) = x * x
square: (x: Int)Int

scala> val xs = res9
xs: List[Int] = List(1, 2, 3)

scala> xs map square
res10: List[Int] = List(1, 4, 9)

scala> xs sum
res11: Int = 6
```



This *is* an expression!
See explanation of map
in following slide

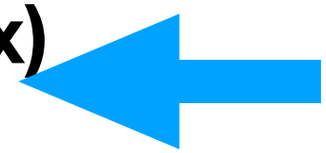
Lists: map

- So, what's this *map* method all about?
- *map* takes an iterable sequence of some sort (a collection) and yields the same sort of sequence except that each element in the result is formed by applying a function to the corresponding element in the original sequence. In the example from the previous slide, *square* is the function.

```
scala> xs map square
res10: List[Int] = List(1, 4, 9)
```

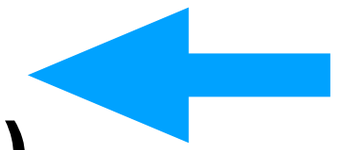
Lists (2)

```
scala> for (x <- xs) yield square(x)
res19: List[Int] = List(1, 4, 9)
```



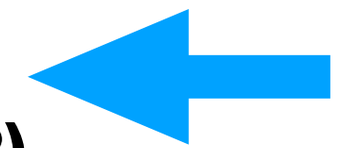
This has the same effect as the map expression above

```
scala> xs :+ 4
res20: List[Int] = List(1, 2, 3, 4)
```



This is also an expression: the original *xs* doesn't change

```
scala> 0 +: xs
res21: List[Int] = List(0, 1, 2, 3)
```



Note that the ":" in an operator associates left or right with the collection

Empty list?

- How can we get an empty list? There are several ways:
  - *Nil*: this is the special name of the empty list—it is a case object that extends the trait *List*.
  - *List():* this is the *apply* method that takes a comma-separated sequence of elements: in this case, none.
  - *List.empty:* this is a standard technique for most collections—if it's possible to have an empty one, then this constant in the companion object will give it.

Lists (3)

- Writing our own *sum* method:

```
scala> def sum(xs: Seq[Int]) = xs match { case Nil => 0; case h :: t => h + sum(t) }
<console>:13: error: recursive method sum needs result type
       def sum(xs: Seq[Int]) = xs match { case Nil => 0; case h :: t => h + sum(t) }
                                                                        ^
scala> def sum(xs: Seq[Int]): Int = xs match { case Nil => 0; case h :: t => h + sum(t) }
sum: (xs: Seq[Int])Int

scala> sum(res20)
res22: Int = 10
```

- Note that we have used *pattern-matching* here. More later

Sum by iteration

- It is of course possible to use a **var** to calculate the sum of a collection:

```
def sumByIteration: Int = {
  var result = 0
  foreach (a => result += a)
  result
}
```

Lists (4)

This will give you your “10 times table” But as a list of 100 Items, not as a table.

```
scala> for (i <- 1 to 10; j <- 1 to 10) yield i * j
val res1: IndexedSeq[Int] = Vector(1,2,3,4,5,6,7,8,9,10,2,4,...
```

This gives an identical result

```
scala> (1 to 10).flatMap(i => (1 to 10).map(j => i * j))
val res2: IndexedSeq[Int] = Vector(1,2,3,4,5,6,7,8,9,10,2,4,...
```

- So, what is this *flatMap* method all about?
- *map* takes an iterable sequence of some sort and yields the same sort of sequence except that each element in the result is formed by applying a function to the corresponding element in the original sequence.

Lists: flatMap

- So, what's this *flatMap* method all about?
- *flatMap* is like *map*, except that the result of the function is not a single element but is a collection of elements (in fact, a collection of the same shape as the original iterable). If we used *map* with the same function, we would end up with a (two-dimensional) table. But *flatMap* flattens the table into a sequence.

```
scala> List(1, 2, 3, 4) flatMap factors
res16: List[Int] = List(1, 1, 2, 1, 3, 1, 2, 4)
```

Map and flatMap

- We will find that these methods, *map* and *flatMap* are really important in functional programming. They crop up everywhere, and not just in sequences or collections.

Infix, postfix, and dot notation

- Let's talk about the syntax of methods
  - *xs.length*: give us the length of some iterable collection *xs*;
    - This (dot notation) follows a Java-like syntax where the dot implies that the method *length* is invoked on the object *xs*. Standard O-O syntax, in other words.
  - *xs length*: is legal but somewhat discouraged.
    - It's called postfix notation and is semantically identical to the dot notation. Indeed, you must *import scala.lang.postfixOps* otherwise you will get a compiler warning.
  - *xs.length()*: is legal but somewhat discouraged unless *length* is invoking a side-effect.
    - It's also discouraged to invoke (or override) a method that has/hasn't parentheses with an invocation (or signature) which hasn't/has parentheses. Stay consistent, in other words.
    - Example: *inputStream.close()*

1-ary methods

- A very common situation is where we have an object (the "receiver"), a method and one parameter to that method, for example:
  - *xs :+ x*
    - ("Infix" notation): this adds an *x* to the right-hand end of collection *xs*.
    - We could also write it as follows: *xs.:+(x)* but we prefer to use the more natural "infix" notation when the method is symbolic (looks like an operator). We generally use the dot notation when the method is verbal.
  - *xs :+ (x, y)*
    - It's even possible to have 2-ary methods written in infix notation.
    - In this context, *(x, y)* is a "tuple" and we might write it this way to add a key-value pair to a map-type object like *HashMap*.
    - Actually, we'd be more likely to write this equivalent form:
      - xs :+ x -> y

Chaining method calls

- There are some situations where it's very typical to use dot notation in a chain.
  - This typically happens with so-called "lens" functions which return a modified version of an object. For example, setting up a Spark session:
    - ```
      val spark: SparkSession = SparkSession
        .builder()
        .appName("WordCount")
        .master("local[*]")
        .getOrCreate()
      ```
- It can also be used when combining *map* methods or other shape-preserving methods on collections, for example:
  - ```
    lines.flatMap(_.split(separator))
    .map((_, 1))
    .reduceByKey(_ + _)
    .sortBy(-_._2)
    ```

Associativity

- Normally, these methods associate to the left, as we'd expect in object-oriented code.

- But Scala also allows us to define methods which associate to the right, by ending the method name with ":"

- For example: *x +: xs* adds the element *x* to head (left-hand end) of the collection *xs*. This can be rewritten in O-O style as *xs.+:(x)* but now it's not so obvious that *x* should end up to the left of *xs*.

- It's easy to remember: in infix notation, the colon (:) always is adjacent to the collection.

Summary

- 1 + 2 is just the more familiar way of writing:
- 1.+(2) which is the object-oriented way of saying that I want to invoke the + method on object 1 (i.e. this = 1) with the parameter (the addend) of 2.
- In Scala there are very few actual operators (maybe none)—everything that looks like an operator is in fact a method on the first of the parameters.