Higher-Order Collection Functions in Scala

§

Made with Obsidian

Type deep-dive | Category computer-science | Technologies Scala, PowerShell 7, VS Code

Website Post Link

Higher-order functions take other functions as arguments and/or return other functions as a result. These powerful abstractions allow us to perform very complex computations with the benefit of reduced verbosity and declaration. Scala has a strong functional component embedded in its architecture; it then makes sense that Scala provides a wide variety of powerful higher-order functions, specifically for collections such as lists, tuples, arrays, vectors, and more.

In this Deep Dive, we'll discuss the main higher-order functions available in Scala, specifically for lists. However, we'll also discuss other collections. We'll go over each one by explaining in detail why they're for, how to use them, and providing various examples, ranging from simpler to more complex ones.

We'll be using Scala worksheets which can be found in the Deep Dive Repo.

§

Table of Contents

- Preparing our environment
- Higher-order functions
    - map
    - flatMap
    - filter
    - foldLeft
    - foldRight
    - reduceLeft
    - reduceRight
    - scanLeft
    - scanRight
    - zip
    - zipWithIndex
    - forall
    - exists
    - groupBy
    - sorted
    - sortBy

    - sortWith
    - minBy & maxBy
    - Creating a generic higher-order collection function
    - Chaining functions
        - mapReduce
        - zipFlattenOrderBy
- Conclusions
- References
- Copyright

§

Preparing our environment

As per usual, we'll prepare a new environment by using `sbt`, VS Code & Metals extension for VS Code.

We'll start by creating a new folder called `higher-order-collection-functions-in-scala`:

Code

```
New-Item -ItemType Directory -Path higher-order-collection-functions-in-scala
cd higher-order-collection-functions-in-scala
```

We'll then create a new Scala project using `sbt`:

Code

```
sbt new sbt new scala/scala3.g8
```

And name it `higher-order-collection-functions`.

We'll open VS Code in our project's root directory, import the current build, and create a new folder & worksheet so that our directory structure looks something as such:

```
├───.bloop
├───.metals
├───.vscode
├───project
├───src
│   ├───main
│   │   └───scala
│   ├───test
│   │   └───scala
│   └───worksheets
│       └───higher-order-list-functions.worksheet.sc
└───target
```

We'll open our `higher-order-list-functions.worksheet.sc` and start exploring.

Higher-order functions

1. map

`map` is a function used to apply a given function to a collection item-wise, meaning each item in our collection is affected by the operation. The return type of `map` is the same as the input type; a new collection with the modified items.

The generic function signature for `map` is the following:

Code

```
def map[A, B](list: List[A])(f: A => B): List[B]
```

Where:

- `A` represents the type of elements in the input list.
- `B` represents the type of elements in the resulting list.
- `list` is the input list on which the `map` operation is performed.
- `f` is the transformation function that maps each element of type `A` to an element of type `B`.
- `List[B]` is the returning list after transformation.

Let us declare a simple list of integer values and apply a square function:

Code

```
val myList1: List[Int] = List(1, 2, 3, 4, 5)
myList1.map(x => x * x)
```

Here, we're declaring our list and deconstructing it using the `x => x * x` pattern. This pattern tells Scala that we're expecting an element `x`, that is, the element to which we're applying the operation, and we're transforming it to be `x` * `x`.

Output

```
res0: List[Int] = List(1, 4, 9, 16, 25)
```

We can also define a more complex function that can be applied to our list, for example, for applying a square operation only for even numbers:

Code

```scala
// A more elaborate function
def applyFun(x: Int): Int =
    if (x % 2 == 0) x * x
    else x

myList1.map(applyFun)
```

Output

```
res1: List[Int] = List(1, 4, 3, 16, 5)
```

Also, we can apply a method for generic `List` types, meaning we're not constrained to the `List[Int]` type:

Code

```scala
// A generic method
def genericMethod[T](x: T) = x match
    case x:Int => x * x
    case x:String => s"${x} * ${x}"

val myList2 = List(1, 2, "3")

myList2.map(genericMethod)
```

Output

```
res2: List[Matchable] = List(1, 4, 3 * 3)
```

However, lists of varying types are not recommended; other collections are more suited for this effect.

What if we have a list of lists and want to sum up all the elements inside each nested list? Well, there are several ways to do so, one being with `map`:

Code

```scala
// A method matching a list of lists
def matchMultiple(xs: List[Int]): Int = xs match
    case Nil => 0
    case y :: ys => y + matchMultiple(ys)

val myList3: List[List[Int]] = List(List(1, 2, 3), List(4, 5, 6), List(7, 8, 9))

myList3.map(matchMultiple)
```

Output

```
res3: List[Int] = List(6, 15, 24)
```

What we're doing here is:

- If the list `xs` is `Nil`, meaning an empty list, return 0.
- If the list `xs` follows the deconstructed pattern `y :: ys`, meaning an item `y` of type `Int` followed by the tail of the list denoted by `ys`, add the item `y` to a recursive call of `matchMultiple` using the tail `ys`.

The map function works here because we're effectively applying a function to each item of the main list `myList3`; in this case, each item is a list itself.

2. flatMap

`flatMap` is similar to `flatten` in that it flattens the collection. We can think of flattening as reducing the dimensions of a given collection. For example, a list of nested lists will be flattened and produce a single list, including all elements inside each nested list.

The generic function signature for `flatMap` is the following:

Code

```
def flatMap[A, B](list: List[A])(f: A => List[B]): List[B]
```

Where:

- `A` represents the type of elements in the input list.
- `B` represents the type of elements in the resulting list.
- `list` is the input list on which the `flatMap` operation is performed.
- `f` is the transformation function that maps each element of type `A` to a list of elements of type `B`.

Let us declare a simple list of lists and apply a recursive function that squares all the nested elements and returns a flattened version of the collection:

Code

```
// Calculate square of nested elements using a recursive function
def squaredSum(xs: List[List[Int]]): List[Int] = xs match
    case Nil => Nil
    case y :: ys => y.map(x => x * x) ++ squaredSum(ys)

val myList4: List[List[Int]] = List(List(1, 2, 3), List(4, 5, 6))

squaredSum(myList4)
```

Output

```
res4: List[Int] = List(1, 4, 9, 16, 25, 36)
```

Let us now perform the same operation using `flatMap`:

Code

```
// Calculate square of nested elements using flatMap & map
myList4.flatMap(x => x.map(x => x * x))
```

Output

```
res5: List[Int] = List(1, 4, 9, 16, 25, 36)
```

We can also simply flatten the collection without any operation:

Code

```
// Return the collection without any operation
myList4.flatMap(x => x)
```

Output

```
res6: List[Int] = List(1, 2, 3, 4, 5, 6)
```

3. filter

`filter` does exactly what it sounds like: It filters elements in a collection by using what's called a predicate. A predicate is a function that evaluates to a Boolean value, depending on whether the condition is met.

The `filter` method is extremely useful since it can take extremely complex predicate functions and can also be combined (*chained*) with other functions to achieve more complex operations.

The generic function signature for `filter` is the following:

Code

```
def filter[A](list: List[A])(predicate: A => Boolean): List[A]
```

Where:

- `(list: List[A])` is the function that takes a parameter named `list`, which is of type `List[A]`.
- `(predicate: A => Boolean)` is the function that takes another parameter named `predicate`, which is a function that takes an element of type `A` and returns a Boolean value.
- `: List[A]` denotes the function's return type.

Let us declare a simple list of integer values and apply a recursive function that verifies if a given element is even:

Code

```
// Declare a recursive function that filters even numbers
def checkEven(xs: List[Int]): List[Int] = xs match
    case Nil => Nil
    case y :: ys => if (y % 2 == 0) y :: checkEven(ys) else checkEven(ys)

val myList7: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
checkEven(myList7)
```

Output

```
res7: List[Int] = List(2, 4, 6, 8)
```

Now, let us perform the exact same operation using `filter`:

Code

```
// Filter even numbers using filter
myList7.filter(x => x % 2 == 0)
```

Output

```
res8: List[Int] = List(2, 4, 6, 8)
```

We can also use `filter` to compare a given condition using the same list elements as reference. For example, we can compare if the entire list is less than the head of the list (*the first element*):

Code

```
// Filter a list of strings
val myList8: List[String] = List("d", "b", "e", "a", "t", "b", "y", "z")

myList8.filter(x => x < myList8.head)
```

Output

```
res9: List[String] = List(b, a, b)
```

If we change this for less than or equal to, the first element will also be included in the resulting collection (*it will filter itself*):

Code

```
myList8.filter(x => x <= myList8.head)
```

Output

```
res10: List[String] = List(d, b, a, b)
```

4. foldLeft

`foldLeft` belongs to a greater classification called `fold`. What `foldLeft` does, is reduce a list from right to left by applying some predefined operation. For example, a fold on `List(1, 2, 3, 4, 5)` will go from right to left, taking each element and applying the operation we provide.

The `fold` methods require what's called an accumulator as a given parameter; this is a value that keeps count of the result of the operation and returns it as a result.

We'll see that the `reduce` methods perform practically the same operation as `fold` and do not require an explicit accumulator declaration. However, using an accumulator different from the first element in our collection can sometimes be advantageous.

The generic function signature for `foldLeft` is the following:

Code

```
def foldLeft[A, B](list: List[A])(initialValue: B)(accumulator: (B, A) => B): B
```

Where:

- `[A, B]` are type parameter declarations. The letter `A` represents the type of elements in the list, and `B` represents the type of the accumulated result.
- `(list: List[A])` is the function that takes a parameter named `list`, which is of type `List[A]`. It represents the input list that will be folded over.
- `(initialValue: B)` is the function that takes another parameter named `initialValue`, which represents the initial value of the accumulator.
- `(accumulator: (B, A) => B)` is the function that takes a third parameter named `accumulator`, which is a function that takes the current accumulated value of type `B` and an element of type `A`, and returns the updated accumulated value of type `B`.
- `: B` denotes the return type of the function. It specifies that the function will return the final accumulated value of type `B`.

Let us provide a simple example where we have a list of lists, and we want to reduce the nested lists by performing a product operation. We can do this using a recursive definition first:

Code

```
// Define a recursive function calculating the product of items and returning a list of integer values
def foldListsLeft(xs: List[List[Int]], f: List[Int] => Int): List[Int] = xs match
    case Nil => Nil
    case y :: ys => f(y) :: foldListsLeft(ys, f)

// Define the product helper function
def productsInt(xs: List[Int]): Int =
    def prodAccumulate(xs: List[Int], acc: Int): Int =
        if (xs.isEmpty) acc
        else xs.head * prodAccumulate(xs.tail, acc)
    prodAccumulate(xs, 1)

val myList9: List[List[Int]] = List(List(1, 2, 3), List(4, 5, 6), List(7, 8, 9))

foldListsLeft(myList9, productsInt)
```

What we're doing is:

- Declaring a recursive function that will go over all nested lists inside the main list.
- Apply a product operation using a helper function `productsInt`
- Calling the recursive function with a list of lists.

Output

```
res11: List[Int] = List(6, 120, 504)
```

The `reduceLeft` method does this, but in three lines:

Code

```
// Calculate sum of products using foldLeft
def foldListsLeft2(xs: List[List[Int]]): List[Int] = xs match
    case Nil => Nil
    case y :: ys => y.foldLeft(1)(_ * _) :: foldListsLeft2(ys)

foldListsLeft2(myList9)
```

Output

```
res12: List[Int] = List(6, 120, 504)
```

What `foldLeft` is doing is:

- Taking the first element of a list of lists (*a list of type* `List[Int]`)
- Folding left by calculating the product of each element with the second one. This is achieved via the deconstruction of `List[Int]` using `_*_`.
- Calling the function recursively by constructing a new list using the `::` operator.

5. foldRight

`foldRight` works the same as foldLeft, the only difference being that the order of operations goes from left to right. This does not matter when performing operations such as product or addition, but it starts to matter when we have less direct operations; even with a subtraction operation, the order starts to matter.

The generic function signature for `foldRight` is the following:

Code

```
def foldRight[A, B](list: List[A])(initialValue: B)(accumulator: (A, B) => B): B
```

Where:

- `[A, B]` are type parameter declarations. The letter `A` represents the type of elements in the list, and `B` represents the type of the accumulated result.
- `(list: List[A])` is the parameter named `list`, which is of type `List[A]`. It represents the input list that will be folded over.
- `(initialValue: B)` is the function that takes another parameter named `initialValue`, which represents the initial value of the accumulator.
- `(accumulator: (A, B) => B)` is the parameter named `accumulator`, which is a function that takes an element of type `A` and the current accumulated value of type `B`, and returns the updated accumulated value of type `B`.
- `: B` denotes the return type of the function. It specifies that the function will return the final accumulated value of type `B`.

Let us define a list of integer values we want to reduce by subtracting all elements. We'll do it both ways: left to right and right to left:

Code

```
// Define a list of integer values
val myList10: List[Int] = List(1, 2, 3, 4)

// Compare between both foldLeft & foldRight
myList10.foldLeft(0)(_ - _)
myList10.foldRight(0)(_ - _)
```

Output

```
res13: Int = -10
res14: Int = -2
```

Here we can see that the order matters since we're dealing with a subtraction operator. This is just a simple use case, but there are more complex cases we can explore. However, we'll leave that to the equivalent `reduce` methods.

6. reduceLeft

As mentioned, `reduceLeft` works practically the same as foldLeft, the main difference being that the `reduce` methods do not require an initial parameter to be explicitly declared.

The generic function signature for `reduceLeft` is the following:

Code

```
def reduceLeft[A](list: List[A])(accumulator: (A, A) => A): A
```

Where:

- `reduceLeft` takes a list of type `A` elements and an accumulator function.
- The accumulator function combines two elements of type `A` and returns a result of type `A`.
- `reduceLeft` applies the accumulator function successively to the list elements from left to right, using the first element as the starting point.
- It returns the final result of type `A`.

Let us define a list of characters we want to concatenate into a single string. We'll first do it by defining a recursive method:

Code

```
def concatList(xs: List[Char]): String = xs match
    case Nil => ""
    case y :: ys => y.toString + concatList(ys)

val myList11: List[Char] = List('S', 'c', 'a', 'l', 'a')

concatList(myList11)
```

Output

```
res15: String = Scala
```

We can start seeing a common pattern here: When we define a recursive method using pattern matching like we've been defining in this segment, we need to declare at least two cases:

- **Case 1:** When the list is empty (`Nil`), return the most basic unit of the return type we're trying to produce.
  - In the case of an `Int` when calculating sums, we need to provide a 0.
  - In the case of an `Int` when calculating products, we need to provide a 1.
  - In the case of a `String` when performing concatenations, we must provide an empty string.
  - In the case of a `Char`, the same applies.
- **Case 2:** When the list is not empty and follows the pattern `y :: ys`, we recursively call our function while performing the desired operation.

We can now perform a similar operation using `reduceLeft`. With `reduceLeft`, it's a little trickier since we cannot directly provide a value of a different type recursively while performing the transformation; we could do that with our recursive declaration since we explicitly defined the function signature to be `concatList(xs: List[Char]): String`.

Because of this, we'll need to use another higher-order function to first change types and then reduce to a string:

Code

```
myList11.map(a => a.toString).reduceLeft(_ + _)
```

Output

```
res16: String = Scala
```

7. reduceRight

`reduceRight` works the same, only that the direction of the reduction is inverted.

The generic function signature for `reduceLeft` is the following:

Code

```
def reduceRight[A](list: List[A])(accumulator: (A, A) => A): A
```

Where:

- `reduceRight` takes a list of type `A` elements and an accumulator function.
- The accumulator function combines two elements of type `A` and returns a result of type `A`.
- `reduceRight` successfully applies the accumulator function to the list elements from right to left, using the last element as the starting point.
- It returns the final result of type `A`.

Contrary to common sense, a reduceRight will not invert our list; instead, it will concatenate in the same order but starting from the inverse direction.

For example, the reduceRight version of our previous implementation would yield the same:

Code

```
myList11.map(a => a.toString).reduceRight(_ + _)
```

Output

```
res17: String = Scala
```

If we want to reverse our list using `reduceRight`, we can exchange each item recursively:

Code

```
val myList12: List[Char] = List('a', 'l', 'a', 'c', 'S')
myList12.map(a => a.toString).reduceRight((a, b) => b + a)
```

Output

```
res18: String = Scala
```

8. scanLeft

`scanLeft` is similar to fold & reduce, but instead of returning an accumulator with the total value computed by a given operation, it introduces a collection of intermediate cumulative results using a start value.

The generic function signature for `scanLeft` is the following:

Code

```
def scanLeft[A, B](list: List[A])(initialValue: B)(accumulator: (B, A) => B): List[B]
```

This is a little confusing, so let's explain it with an example. Let us declare a simple list of integer values and apply a `scanLeft` method using the addition operator:

Code

```
val myList13: List[Int] = List(2, 2, 2, 2, 2, 2, 2)

myList13.scanLeft(2)(_ * _)
```

Output

```
res19: List[Int] = List(2, 4, 8, 16, 32, 64, 128, 256)
```

This operation returns the power sequence of 2. Interesting. So, in general terms, the `scanLeft` function goes over each element in our list, starting from a predefined accumulator, computing the product operation between a pair of elements, and returning a collection with the accumulated results on each step.

We can visualize each step of the `reduceLeft` method in this diagram:

```
2 = 2
2 * 2 = 4
2 * 4 = 8
2 * 8*
...
```

Where the first element is our initial accumulator parameter.

If we modify our initial accumulator to 1, the behavior is the same, with the difference of the first element:

Code

```
myList13.scanLeft(1)(_ * _)
```

Output

```
res20: List[Int] = List(1, 2, 4, 8, 16, 32, 64, 128)
```

So the accumulator simply serves to mark the first entry, and from there, each accumulation is recursively inserted in our new sequence.

9. scanRight

`scanRight` is the scanLeft equivalent in the inverse direction, meaning the order of items in the resulting collection will be inverse of `reduceLeft`.

The generic function signature for `scanRight` is the following:

Code

```
def scanRight[A, B](list: List[A])(initialValue: B)(accumulator: (A, B) => B): List[B]
```

Where:

- `scanRight` takes a list of elements of type `A`, an initial value of type `B`, and an accumulator function.
- The accumulator function combines an element of type `A` with the accumulated value of type `B` and returns an updated value of type `B`.
- `scanRight` applies the accumulator function successively to the list elements from right to left, starting with the initial value.
- It returns a new list of accumulated values, where each element represents the result of applying the accumulator function up to that point. The type of the elements in the resulting list is `B`.

We can use `reduceRight` to calculate the power of a given integer number:

Code

```
def computePower(x: Int, y: Int): Int =
    val myCollection: List[Int] = List.fill(y)(x)
    myCollection.scanRight(1)(_ * _).head

computePower(2, 3)
```

What `computePower` is doing is:

- It accepts the integer number `x` elevated to the power of `y`.
- It then creates a list containing `x` elements `y` times.
- Finally, it uses `scanRight` with the initial accumulator defined as `1`, and a product operation throughout the entire collection.
- It returns the head of the collection, which will be the power of the number elevated to `y`.

Output

```
res21: Int = 8
```

10. zip

We might have heard of the `zip` function from a Python context, for example, when trying to create dictionaries. The `zip` method in Scala is used to merge a collection with a current collection resulting in a new collection of pairs of tuple elements from both collections.

Essentially, we're packing two collections into one, creating combinations of collection A & collection B.

The generic function signature for `scanRight` is the following:

Code

```
def zip[A, B](list1: List[A], list2: List[B]): List[(A, B)]
```

Where:

- `zip` takes two lists, `list1` of type `List[A]` and `list2` of type `List[B]`.
- It combines the corresponding elements from both lists and returns a new list where each element is a tuple `(A, B)` representing the pairs of elements from the original lists.
- The resulting list will have a length equal to the shorter of the two input lists.
- The type of elements in the resulting list is `(A, B)`, a tuple where the first element comes from `list1` and the second element comes from `list2`.

Let us start by declaring a recursive method to accomplish a `zip` operation:

Code

```
def zipCollections(xs: List[Int], ys: List[Int]): List[List[Int]] = (xs, ys) match
    case (Nil, Nil) => Nil
    case (Nil, z :: zs) => Nil
    case (z :: zs, Nil) => Nil
    case (z :: zs, d :: ds) => List(z, d) :: zipCollections(zs, ds)

val myList14: List[Int] = List(1, 2, 3, 4)
val myList15: List[Int] = List(4, 3, 2, 1)

zipCollections(myList14, myList15)
```

Output

```
res22: List[List[Int]] = List(List(1, 4), List(2, 3), List(3, 2), List(4, 1))
```

What we're doing here is covering all possible cases that can occur in a pair of lists of integers by matching our two lists simultaneously. This can be done because we're packing our pair of lists in a tuple; we then deconstruct this tuple into elements `xs` and `ys` in the cases below.

This ensures that if a list `A` has a different length than a list `B`, the resulting pair will be `Nil`:

Code

```
val myList14: List[Int] = List(1, 2, 3, 4)
val myList15: List[Int] = List(4, 3, 2)
```

Output

```
res22: List[List[Int]] = List(List(1, 4), List(2, 3), List(3, 2))
```

We can do the same in a single line by using the `zip` method:

Code

```
myList14.zip(myList15)
```

Output

```
res23: List[Tuple2[Int, Int]] = List((1,4), (2,3), (3,2), (4,1))
```

11. zipWithIndex

`zipWithIndex` is similar to `zip`, the difference being that instead of combining elements from two separate lists, we include an index to the first element of a list `A`.

The generic function signature for `zipWithIndex` is the following:

Code

```
def zipWithIndex[A](list: List[A]): List[(A, Int)]
```

Where:

- `zipWithIndex` is a function in Scala that takes a list of elements of type `A`.
- It pairs each element of the list with its corresponding index.
- The resulting list will contain tuples `(A, Int)`, where the first element is an element from the original list, and the second element is its corresponding index.
- The index starts from 0 for the first element and increments by 1 for each subsequent element.

Let us declare a list of strings containing random names and apply the `zipWithIndex` function to see what it does:

Code

```
val myList16: List[String] = List("Carolina", "Emma", "Diego", "Will", "Charles")
myList16.zipWithIndex(0)
```

Output

```
res24: Tuple2[String, Int] = (Carolina,0)
```

So, the index we specify is applied to the first element of the list. This is not useful for one element, but we can iterate over the entire collection and assign an incremental index to our whole collection. There are two main easy methods we can use to approach this:

- By using the `foreach` iterator.
- By using `map`.

Code

```
// Using map
myList16.zipWithIndex.map(x => x)

// Using foreach
myList16.zipWithIndex.foreach(x => println(x))
```

Output

```
res25: List[Tuple2[String, Int]] = List((Carolina,0), (Emma,1), (Diego,2), (Will,3), (Charles,4))

// (Carolina,0)
// (Emma,1)
// (Diego,2)
// (Will,3)
// (Charles,4)
```

The results will be of a different type since, on the first one, we're applying the function whose return value can be assigned to a new variable, while on the second one, the return type of `foreach` will always be `Unit`; in short, if we want to create a new collection with indexed elements, we use `map` or `flatMap`.

12. forall

`forall` is used to evaluate if all elements in a given collection comply with a specified predicate. As we have seen, a predicate is a function that accepts a "*test*" and returns a `Boolean` value depending on the outcome of the test.

As we will see, `forall` is similar to `filter` in this way, but the difference is that the first one returns a `Boolean` value if the entire collection complies. In contrast, the second one returns the element that complied with the predicate.

The generic function signature for `forall` is the following:

Code

```
def forall[A](list: List[A])(predicate: A => Boolean): Boolean
```

Where:

- `forall` takes a list of elements of type `A` and a predicate function `predicate`.
- The predicate function takes an element of type `A` and returns a Boolean value.
- `forall` applies the predicate function to each element in the list and checks if the predicate holds for all elements.

- It returns a `Boolean` value indicating whether the predicate holds for all elements in the list.
- The return type is `Boolean`. It will be `true` if the predicate holds for all elements and `false` otherwise.

We can check if a list contains exclusively even integers:

Code

```
val myList17: List[Int] = List(1, 2, 3, 4, 5)
val myList18: List[Int] = List(2, 4, 6, 8)

myList17.forall(x => x % 2 == 0)
myList18.forall(x => x % 2 == 0)
```

Output

```
res27: Boolean = false
res28: Boolean = true
```

We can implement a similar definition by using a recursive declaration, leveraging the logical operator `&`:

Code

```
// Implementing forall using a recursive function
def compliesAll(xs: List[Int]): Boolean = xs match
    case Nil => throw Error("Nothing")
    case y :: Nil => (y % 2 == 0)
    case y :: ys => (y % 2 == 0) & compliesAll(ys)

compliesAll(List(2, 2, 1, 3))
compliesAll(List(2, 2, 4, 6))
compliesAll(List(1, 2, 2))
compliesAll(List(1))
compliesAll(List(0))
compliesAll(List())
```

Output

```
res29: Boolean = false
res30: Boolean = true
res31: Boolean = false
res32: Boolean = false
res33: Boolean = true
java.lang.Error: Nothing
```

What we're doing here is the following:

- We first check if the list is empty and return `Nil` if it is.
- We then check if the list has one non-empty element, `y`, followed by an empty element `Nil`. In that case, we only check the truth of the element `y`.
- Finally, we check for the list with at least two non-empty elements `y :: ys`. We must note that `ys` is not necessarily a non-empty element. In fact, it can be non-empty, but the condition above checks for that

specific case; if we remove the pattern `y :: Nil`, the evaluation will always result in `Error("Nothing")`, since, in the end, an empty list is guaranteed to always be returned as the last argument in our evaluation.

In this implementation, we're leveraging the use of the logical operator `&` (*if there is any `Boolean` value that evaluates to `false`, the resulting value will be false, no matter what*)

We can also perform this same implementation by using a non-recursive option, where we combine `map` & `flatten`; the idea about functional programming is that there are always many different ways to tackle a problem; it's up to us to decide which implementation is the most efficient, less resource-intensive, and cleaner in terms of syntaxis.

13. exists

The `exists` method is the `OR` version of the above; it evaluates all elements and returns `true` if at least one element complies with the provided predicate.

In fact, if we look at the function signature, it's exactly the same as `forall`. The generic function signature for `exists` is the following:

Code

```
def exists[A](list: List[A])(predicate: A => Boolean): Boolean
```

Where:

- `exists` takes a list of elements of type `A` and a predicate function `predicate`.
- The predicate function takes an element of type `A` and returns a Boolean value.
- `exists` applies the predicate function to each element in the list and checks if the predicate holds for at least one element.
- It returns a `Boolean` value indicating whether the predicate holds for at least one element in the list.
- The return type is `Boolean`. It will be `true` if the predicate holds for at least one element and `false` if the predicate holds false for all elements.

Now that we have the recursive implementation for `forall`, it's fairly straightforward to define one for `exists`:

Code

```
// Implementing exists using a recursive function
def compliesOne(xs: List[Int]): Boolean = xs match
    case Nil => throw Error("Nothing")
    case y :: Nil => (y % 2 == 0)
    case y :: ys => (y % 2 == 0) | compliesOne(ys)

compliesOne(List(1, 3, 5, 7, 9))
compliesOne(List(2, 2, 4, 6))
compliesOne(List(1, 2, 2))
compliesOne(List(1))
compliesOne(List(0))
compliesOne(List())
```

Output

```
res35: Boolean = false
res36: Boolean = true
res37: Boolean = true
res38: Boolean = false
res39: Boolean = true
java.lang.Error: Nothing
```

We can see that the only modification is the logical operator; we change `&` for `|`.

Similar to the example above, we can check if a list contains at least one even integer:

Code

```scala
val myList20: List[Int] = List(1, 3, 5, 7, 9)
val myList21: List[Int] = List(2, 4, 6, 8)

myList20.exists(x => x % 2 == 0)
myList21.exists(x => x % 2 == 0)
```

Output

```
res40: Boolean = false
res41: Boolean = true
```

14. groupBy

The `groupBy` method partitions a collection into a `HashMap` according to some discriminator function. This function can, for example, evaluate if a given subset of the list contains an even number and group by different columns depending on the truth of the evaluation.

The generic function signature for `groupBy` is the following:

Code

```scala
def groupBy[A, B](list: List[A])(keyFunction: A => B): Map[B, List[A]]
```

Where:

- `groupBy` takes a list of elements of type `A` and a key function `keyFunction`.
- The key function maps each element of type `A` to a key of type `B`.
- `groupBy` groups the list elements based on the keys obtained from the key function.
- It returns a `Map` where the keys are of type `B`, and the values are lists of type `A` elements with the same key.
- The return type is `Map[B, List[A]]`, representing a mapping from keys of type `B` to lists of type `A` elements.

This one might seem slightly more confusing; we can illustrate how it works with a simple use case:

Let us define a list that contains odd & even numbers. We want to group our list into two different mappings: the first one will contain the even numbers, and the second one the odd ones:

Code

```
// Define a discriminator function
def groupNumbers(x: Int): Int =
    if (x % 2 == 0) 1
    else 2

val myList22: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)

// Group by using discriminator function
myList22.groupBy(groupNumbers)
```

Output

```
res42: Map[Int, List[Int]] = HashMap(1 -> List(2, 4, 6, 8), 2 -> List(1, 3, 5, 7, 9))
```

As we can see, the result is a `HashMap` structure that maps each key with its corresponding subset based on the discriminator function `groupNumbers`.

As we can imagine, this method is extremely powerful since it lets us compose infinitely complex discriminators while returning the very fast `HashMap` data structure.

Of course, keys can be of different types than the grouped subsets:

Code

```
// Define a discriminator function now returning a string as key
def groupNumbersStringKey(x: Int): String =
    if (x % 2 == 0) "Even"
    else "Odd"

// Group by using discriminator function
myList22.groupBy(groupNumbersStringKey)
```

Output

```
res43: Map[String, List[Int]] = HashMap(Odd -> List(1, 3, 5, 7, 9), Even -> List(2, 4, 6, 8))
```

If we want to assign our `groupBy` result to a variable and then index our `HashMap` structure to return a given subset, we can do so:

Code

```
// Group by using discriminator function
val groupedInts = myList22.groupBy(groupNumbersStringKey)
groupedInts("Odd")
```

Output

```
res43: List[Int] = List(1, 3, 5, 7, 9)
```

15. sorted

`sorted` is used for sorting collections. The sort is stable, meaning that equal elements (*as determined by* *`ord.compare`*) appear in the same order in the sorted sequence as in the original.

The generic function signature for `sort` is the following:

Code

```
def sort[A](list: List[A])(implicit ord: Ordering[A]): List[A]
```

Where:

- The `sort` function takes a list of elements of type `A` and an implicit `Ordering` instance for type `A`.
- It sorts the elements of the list in ascending order according to the natural ordering of type `A`.
- The return type is `List[A]`, representing the sorted list.

Sorted can be called without parameters or with an `ord` parameter included:

Code

```
val myList23: List[Int] = List(5, 2, 3, 1, 4)

// Calling sorted without parameters
myList23.sorted

// Calling sorted with ord parameter
myList23.sorted(Ordering.Int.reverse)
```

Output

```
res44: List[Int] = List(1, 2, 3, 4, 5)
res45: List[Int] = List(5, 4, 3, 2, 1)
```

As we can see from the generic function signature, the `ord` parameter requires an `Ordering[B]` type. `Ordering` is a trait whose instances represent a strategy for sorting instances of a type. In our case, we used `reverse`, but we can also use custom functions.

16. sortBy

sortBy is similar to `sorted` in that it sorts a collection. The difference is that sortBy accepts sorting using multiple attributes of the elements inside the collection. This is not super useful if we only have a list of integers, but it becomes interesting when we have collections of objects.

The generic function signature for `sortBy` is the following:

Code

```scala
def sortBy[A, B](list: List[A])(f: A => B)(implicit ord: Ordering[B]): List[A]
```

Where:

- The `sortBy` function takes a list of elements of type `A`, a function `f` that maps elements of type `A` to keys of type `B`, and an implicit `Ordering` instance for type `B`.
- It sorts the list elements based on the keys obtained from applying the function `f` to each element.
- The return type is `List[A]`, representing the sorted list.

Let us exemplify by creating a collection of objects, including some simple attributes we can use to sort our collection:

Code

```scala
// Declare a Doggo class
case class Doggo(name: String, age: Int, city: String, owner: String, breed: String)

// Create some instances
val Tommy: Doggo = new Doggo("Tommy", 12, "Budapest", "Laszlo", "Borzoi")
val Borys: Doggo = new Doggo("Borys", 10, "Warsaw", "Bartosz", "Husky")
val Ramon: Doggo = new Doggo("Ramon", 15, "San Juan", "Alondra", "Labradoodle")
val Fumiko: Doggo = new Doggo("Fumiko", 15, "Tokyo", "Keiko", "Shiba Inu")

// Create a list of Doggos
val listOfDoggos: List[Doggo] = List(Tommy, Borys, Ramon, Fumiko)
println(listOfDoggos)
```

Output

```
// List(Doggo(Tommy,12,Budapest,Laszlo,Borzoi), Doggo(Borys,10,Warsaw,Bartosz,Husky),
Doggo(Ramon,15,San Juan,Alondra,Labradoodle), Doggo(Fumiko,15,Tokyo,Keiko,Shiba Inu))
```

As we can see, we have a total of 5 attributes:

- Four `String` type attributes
- One `Int` type attribute

We can then sort by one or more attributes using `sortBy`:

Code

```scala
// Sort by one attribute
listOfDoggos.sortBy(_.name)

// Sort by two attributes
listOfDoggos.sortBy(doggo => (doggo.age, doggo.name))
```

Output

```
res47: List[Doggo] = List(Doggo(Borys,10,Warsaw,Bartosz,Husky), Doggo(Fumiko,15,Tokyo,Keiko,Shiba
Inu), Doggo(Ramon,15,San Juan,Alondra,Labradoodle), Doggo(Tommy,12,Budapest,Laszlo,Borzoi))

res48: List[Doggo] = List(Doggo(Borys,10,Warsaw,Bartosz,Husky),
Doggo(Tommy,12,Budapest,Laszlo,Borzoi), Doggo(Fumiko,15,Tokyo,Keiko,Shiba Inu),
Doggo(Ramon,15,San Juan,Alondra,Labradoodle))
```

In `res48`, we have two doggos of the same age, and the sorting is ultimately resolved using the doggos' names.

We can imagine that we can use whichever object we can think of and include several attributes to sort by.

17. sortWith

`sortWith` also sorts elements in a collection. The extra feature here is that we can use a custom function to sort.

The generic function signature for `sortWith` is the following:

Code

```scala
def sortWith[A](list: List[A])(comparator: (A, A) => Boolean): List[A]
```

Where:

- The `sortWith` function takes a list of elements of type `A` and a binary comparison function `comparator`.
- It sorts the list elements based on the comparison function `comparator`.
- The comparison function should return `true` if the first element should be ordered before the second element and `false` otherwise.
- The return type is `List[A]`, representing the sorted list.

We can build on our previous example and declare a sorting function that will define how noisy a doggo is, based on its breed:

Code

```scala
def noiseLevel(breed: String): Int = breed match
    case "Husky" => 10
    case "Shiba Inu" => 8
    case "Labradoodle" => 4
    case "Borzoi" => 3

listOfDoggos.sortWith((x, y) => (noiseLevel(x.breed) > noiseLevel(y.breed)))
```

And to no surprise, Borys turns out to be the first place in the list:

Output

```
res49: List[Doggo] = List(Doggo(Borys,10,Warsaw,Bartosz,Husky), Doggo(Fumiko,15,Tokyo,Keiko,Shiba
Inu), Doggo(Ramon,15,San Juan,Alondra,Labradoodle), Doggo(Tommy,12,Budapest,Laszlo,Borzoi))
```

18. minBy & maxBy

`minBy` & `maxBy` are equivalent functions: Both take a predicate function as their parameter and apply it to every element in the collection to return the smallest/biggest element:

The generic function signature for `minBy` is the following:

Code

```
def minBy[A, B](list: List[A])(f: A => B)(implicit ord: Ordering[B]): A
```

Where:

- The `minBy` function takes a list of elements of type `A`, a function `f` that maps elements of type `A` to keys of type `B`, and an implicit `Ordering` instance for type `B`.
- It finds the minimum element from the list based on the keys obtained from applying the function `f` to each element.
- The return type is `A`, representing the minimum element.
- `A` represents the type of elements in the list.
- `B` represents the type of keys obtained from the key function or mapping function.
- `ord: Ordering[T]` is an implicit parameter that provides an instance of the `Ordering` type class for type `T`.
- The implicit `Ordering` instance defines the ordering of elements when sorting or finding minimum/maximum values.

While the generic function signature for `maxBy` is the following:

Code

```
def maxBy[A, B](list: List[A])(f: A => B)(implicit ord: Ordering[B]): A
```

Where:

- The `maxBy` function takes a list of elements of type `A`, a function `f` that maps elements of type `A` to keys of type `B`, and an implicit `Ordering` instance for type `B`.
- It finds the maximum element from the list based on the keys obtained from applying the function `f` to each element.
- The return type is `A`, representing the maximum element.
- `A` represents the type of elements in the list.
- `B` represents the type of keys obtained from the key function or mapping function.
- `ord: Ordering[T]` is an implicit parameter that provides an instance of the `Ordering` type class for type `T`.
- The implicit `Ordering` instance defines the ordering of elements when sorting or finding minimum/maximum values.

Let us exemplify both by using our previous example, where we want to get the loudest and quietest doggos in the pack, the first one to counterattack our neighbor's lawn mower at 6 am, and the latter to keep ourselves company by the fireplace while reading Tolstoy:

Code

```
listOfDoggos.minBy(x => noiseLevel(x.breed))
listOfDoggos.maxBy(x => noiseLevel(x.breed))
```

Output

```
res50: Doggo = Doggo(Tommy,12,Budapest,Laszlo,Borzoi)
res51: Doggo = Doggo(Borys,10,Warsaw,Bartosz,Husky)
```

Nah, Borszois are not my cup of tea. I'll stick with Borys. Good luck neighbor.

§

Creating a generic higher-order collection function

Until now, we've worked with fixed types in our recursive function declarations, meaning the function expects, for example, a list of integer values, chars, a list of strings, etc.

The power of defining our methods is that we can abstract a specific type function into a generic type. This is achieved via the `T` parameter declaration.

Let us explore a case where we have a list of unknown type, and we'd like to perform the same operation without having to explicitly declare our list as `List[Int]`, for example:

Code

```
def genericMap[T](xs: List[T], f: T => T): List[T] = xs match
    case Nil => Nil
    case y :: ys => f(y) :: genericMap(ys, f)

val myListInts: List[Int] = List(1, 2, 3, 4)
val myListStrings: List[String] = List("1", "2", "3", "4")
val myListDoubles: List[Double] = List(1.0, 2.0, 3.0, 4.0)

genericMap(myListInts, x => x * 2)
genericMap(myListStrings, x => x * 2)
genericMap(myListDoubles, x => x * 2)
```

Output

```
res24: List[Int] = List(2, 4, 6, 8)
res25: List[String] = List(11, 22, 33, 44)
res26: List[Double] = List(2.0, 4.0, 6.0, 8.0)
```

What we're doing is specifying our function in terms of a type parameter or type variable denoted by `T`. This parameter acts as a placeholder for a specific type that will be determined when the function is invoked, or the generic class is instantiated and is what lets us declare generic functions that work with different types.

However, there's a tradeoff; we must ensure that the `map` function we're defining, in this case, `x * 2`, is compatible with the types we're specifying. If this is not the case, we'll need to define separate methods depending on the type used; this can be achieved via additional pattern matching, for example.

§

Chaining functions

Functional programming encourages what's called the chaining of operations. This means we can apply one function after another using a single line without requiring separate variable definitions for each step. Chaining works because the evaluation result of a given function `A` is taken as an argument of the subsequent function `B`, and so on.

This might sound similar to what SQL does, and that's because they use the same underlying concept called "*fluent API*" or "*method chaining*".

Fluent API is a design pattern that enables code to be written in a way that resembles natural language or a domain-specific language (*DSL*). It allows for chaining multiple operations or method calls together concisely and readably.

Both Scala and SQL share this characteristic because they have adopted the fluent API pattern in their respective designs.

While method chaining is extremely powerful, it must be used carefully since its objective is to simplify syntax & improve readability; overusing this feature might result in illegible code.

1. mapReduce

Let us start with a simple example where we want to apply two functions on a list of lists:

- First, calculate the square of all the elements contained in the nested lists
- Then, perform a sum operation for all the nested elements.

Code

```
// map reduce
def mapReduce(xs: List[List[Int]]): List[Int] = xs match
    case Nil => Nil
    case y :: ys => y.map(x => x * x).reduce(_ + _) :: mapReduce(ys)

val myList5: List[List[Int]] = List(List(1, 2, 3), List(4, 5, 6), List(7, 8, 9))

mapReduce(myList5)
```

Output

```
res7: List[Int] = List(14, 77, 194)
```

Here, we're chaining two functions: `map` & `reduce`, where:

- `map` applies a square operation to each element contained in the nested lists.

- `reduce` reduces all elements on each nested list to a single number by performing a sum operation.

2. zipFlattenOrderBy

Suppose we have two individual lists of type `List[String]`; the first one contains the first names of iconic 19th-century writers, while the second one contains the writers' surnames in the same order, meaning `List[A](1)` will belong to `List[b](1)`, and so on.

We're given a list of lists so that list `C` has the type `List[List[String]]`. Our task is to transform this collection so that we end up with a name surname association with the type `List[Tuple]` ordered alphabetically ascending by name, where each tuple represents a combination of name and surname for a given author.

We can use pattern matching to deconstruct our initial collection and then use the following:

- `flatten` to flatten the tail of our list, in this case, the surnames.
- `zip` to glue both lists together, resulting in the required collection type, `List[Tuple]`.
- `sortBy` to sort by the first name.

Code

```
// zip + flatten + order by
def zipFlattenOrderBy(x: List[List[String]]): List[Tuple] = x match
    case Nil => Nil
    case y :: ys => y.zip(ys.flatten(x => x)).sortBy(_._1)

val writerNames: List[String] = List("Emily", "Leo", "Charles", "Oscar", "Alexander", "Henry")
val writerSurnames: List[String] = List("Brontë", "Tolstoy", "Dickens", "Wilde", "Pushkin",
"James")

val writerList: List[List[String]] = List(writerNames, writerSurnames)

zipFlattenOrderBy(writerList)
```

And as expected, we end up with the required collection sorted by first name:

Output

```
res56: List[Tuple] = List((Alexander,Pushkin), (Charles,Dickens), (Emily,Brontë), (Henry,James),
(Leo,Tolstoy), (Oscar,Wilde))
```

Of course, we could have used both separate lists and save ourselves the flattening step, but this is just to illustrate that higher-order collection methods can be used in all kinds of forms; from chaining to using as arguments for other functions.

§

Conclusions

We've reviewed the most important higher-order collection functions available in Scala while mentioning their generic signature, discussing some of their most important traits, and implementing recursive alternatives using pattern matching that helped us exemplify more clearly how they work. We also presented some simple examples that hopefully helped clear things up. Lastly, we went over an example where we created our higher-order collection function variation using type variables, and we also discussed how we could chain one or more functions to build more complex transformation queries.

One of Scala's specialties is working with higher-order functions; a great deal of the language is implemented using them. However, we must remember that they are not always the best or most efficient approach; they are extremely powerful, but sometimes we can use other methods, such as for-expressions, for a clearer and more concise syntax.

§

References

- Higher-Order Functions, Scala
- Functional Programming Principles in Scala, Coursera / EPFL

§

Copyright