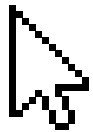
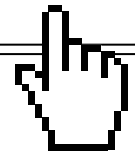
Higher Order Functions

Currying, Collections and Callbacks, and Abstract Data Types



Valeria Saldana



01 Currying

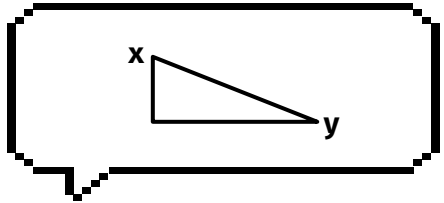
How and why it is used?

02 Collections and Callbacks

Map, FlatMap, FoldRight, and Other Folds and Reduce

03 Abstract Data Types

Map, Set, Parallel and Distributed

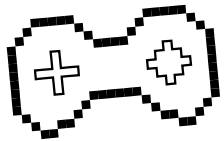


01

Currying

Currying

Definition: The process of transforming a function with multiple variables into multiple functions, each taking a single argument.



Transformation:

- A function f(x, y) that takes two arguments can be transformed into f(x)(y).
- The first function takes x and returns a new function that takes y.

Example

```
def add(x: Int, y: Int): Int = x + y   // Regular function with two arguments

def addCurried(x: Int)(y: Int): Int = x + y  // Curried version
```

How it works:

- Call addCurried(3) → This returns a function (y: Int) ⇒ 3 + y.
- Call that returned function with 4 → addCurried(3)(4) results in 7.

Benefits

Peartial Applications

Allows fixing some arguments and reusing the resulting function.

```
val addThree = addCurried(3) // Partially
applied function
println(addThree(4)) // Outputs: 7
```

Key Concept

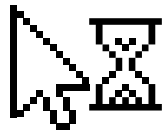
Currying turns a function into a chain of functions, each handling one argument at a time.

Exercise

Suppose you are tasked with implementing a curried function in Scala that performs the following:

1. Takes a String prefix as its first argument.
2. Takes an Int multiplier as its second argument.
3. Takes a list of numbers as its third argument.
4. Returns a new list where each number is multiplied by the given multiplier, converted to a String, and prefixed with the given prefix.

show an example where you fix the prefix and multiplier using partial application, then process multiple lists of numbers.



01 Define the Curried Function:

```
def processList(prefix: String)(multiplier: Int)(numbers: List[Int]): List[String] = {
  numbers.map(n ⇒ prefix + (n * multiplier).toString)
}
```

02 Partial Application:

```
val prefixWithMultiplier = processList("Result: ")(2) // Fix prefix as "Result: " and
multiplier as 2
```

03 Define lists of numbers:

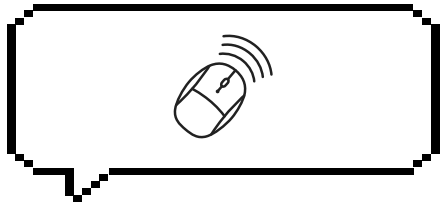
```
val list1 = List(1, 2, 3)
val list2 = List(4, 5, 6)
```

04 Apply the partially applied function to lists:

```
val result1 = prefixWithMultiplier(list1)
val result2 = prefixWithMultiplier(list2)
```

05 Print:

```
println(result1) // Output: List("Result: 2", "Result: 4", "Result: 6")
println(result2) // Output: List("Result: 8", "Result: 10", "Result: 12")
```



02

Collections and Callbacks

Maps

Purpose: Transforms each element of a collection into a new element using a provided callback function.

Signature:

```
def map[B](f: A ⇒ B): List[B]
```

- Takes a function f that maps each element A to B.
- Returns a new collection containing the transformed elements.

Example:

```
val numbers = List(1, 2, 3)
val doubled = numbers.map(x ⇒ x * 2) // List(2, 4, 6)
println(doubled)
```

flatMap

Purpose: Similar to map, but the callback function returns a collection rather than a single element. flatMap flattens the resulting collections into a single collection.

Signature:

```
def flatMap[B](f: A ⇒ List[B]): List[B]
```

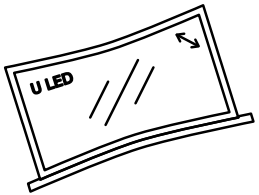
- Use Case: When each element can be mapped to zero or more elements.

Example:

```
val numbers = List(1, 2, 3)
val expanded = numbers.flatMap(x ⇒ List(x, x * 10))
println(expanded) // List(1, 10, 2, 20, 3, 30)
```

`x ⇒ List(x, x * 10)` maps each number to a list of two elements, and flatMap combines all these lists into a single list.

foldRight



Purpose: Combines elements of a collection using a binary operation, starting from the rightmost element.

Signature:

```
def foldRight[B](z: B)(op: (A, B) ⇒ B): B
```

- z is the starting value (accumulator).
- op is a binary operation: it combines each element with the current accumulator.

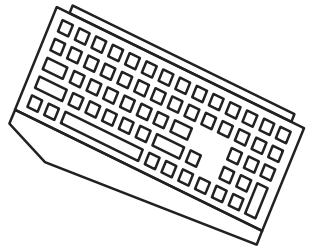
Right-to-Left Processing: The operation starts from the end of the list and moves to the beginning.

Example:

```
val numbers = List(1, 2, 3)
val sum = numbers.foldRight(0)((x, acc) ⇒ x + acc) // 1
+ (2 + (3 + 0))
println(sum) // Output: 6
```

foldLeft

Purpose: Similar to foldRight, but processes elements left-to-right (from the start of the list).



Signature:

```
def foldLeft[B](z: B)(op: (B, A) ⇒ B): B
```

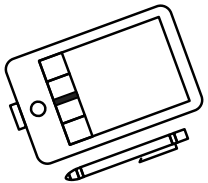
Example:

```
val numbers = List(1, 2, 3)
val sum = numbers.foldLeft(0)((acc, x) ⇒ acc + x) //
((0 + 1) + 2) + 3
println(sum) // Output: 6
```

Key Difference:

- foldLeft: Left-to-right, accumulator is updated first.
- foldRight: Right-to-left, last element is processed first.

Reduce



Purpose: Combines all elements of a collection using a binary operation, without an initial value.

Signature:

```
def reduce(op: (A, A) ⇒ A): A
```

Key Point: Unlike fold, reduce requires the collection to be non-empty because there is no initial value (z).

Example:

```
val numbers = List(1, 2, 3)
val sum = numbers.reduce((x, y) ⇒ x + y) // 1 + 2 + 3
println(sum) // Output: 6
```

When to use each?

- Use `map` when transforming elements directly.
- Use `flatMap` when each element results in a collection.
- Use `foldLeft` or `foldRight` for accumulation with an initial value.
- Use `reduce` for accumulation when no initial value is needed.

Exercise

You have a list of students, where each student is represented as a case class:

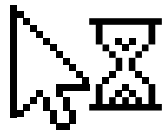
```
case class Student(name: String, scores: List[Int])
```

For example:

```
val students = List( Student("Alice", List(85, 90, 78)), Student("Bob", List(92, 88, 95)), Student("Charlie", List(70, 65, 80)) )
```

Your tasks are:

Task 1: Use map to create a list of students' average scores as a tuple: (name, averageScore).
Task 2: Use flatMap to generate a flat list of all scores.
Task 3: Use foldLeft to calculate the total sum of all scores in the class.
Task 4: Use reduce to find the student with the highest average score.



01 Define the Data and Case Class:

```
case class Student(name: String, scores: List[Int])
val students = List(
    Student("Adrian", List(85, 90, 78)),
    Student("Bob", List(92, 88, 95)),
    Student("Charlie", List(70, 65, 80))
)
```

02 Use map to Get (Name, Average Score):

```
val nameAndAverage = students.map(student ⇒ {
    val average = student.scores.sum.toDouble / student.scores.size
    (student.name, average)
})
println(nameAndAverage)// Output: List((Adrian,84.33333333333333),
(Bob,91.66666666666667), (Charlie,71.66666666666667))
```

03 Use flatMap to Get a Flat List of All Scores:

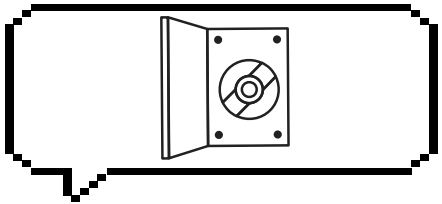
```
val allScores = students.flatMap(_.scores)
println(allScores)
// Output: List(85, 90, 78, 92, 88, 95, 70, 65, 80)
```

04 Use foldLeft to Calculate the Total Sum of All Scores

```
val totalScore = allScores.foldLeft(0)((acc, score) ⇒ acc + score)
println(totalScore)
// Output: 743
```

05 Use reduce to Find the Student with the Highest Average Score:
First, calculate the averages for all students, and then use reduce to compare their averages.

```
val topStudent = nameAndAverage.reduce((student1, student2) ⇒
    if (student1._2 > student2._2) student1 else student2
)
println(topStudent)
// Output: (Bob,91.66666666666667)
```



03

Abstract Data Types

Sets

Definition: A Set is a collection of unique elements with no duplicates.

Key Characteristics:

- Commonly used for membership testing, union, intersection, and difference operations.
- Immutable sets are more common in functional programming, ensuring immutability guarantees.

Key Operations:

- set.contains(elem): Checks if the element exists in the set.
- set + elem: Returns a new set with the element added.
- set - elem: Returns a new set with the element removed.
- Set Operations: union, intersection, and difference.

Example

```
val set1 = Set(1, 2, 3)
val set2 = Set(3, 4, 5)

val union = set1.union(set2)
// Set(1, 2, 3, 4, 5)
val intersection =
set1.intersect(set2) //
Set(3)
val difference =
set1.diff(set2) // Set(1, 2)

println(union)
println(intersection)
println(difference)
```

Parallel and Distributed Collections

Definition: Parallel and distributed collections are used to process large datasets efficiently by dividing work across multiple CPU cores or distributed systems.

Key Characteristics:

- Operations like map, reduce, and fold are executed in parallel or distributed across nodes.
- Functional programming provides guarantees like immutability, ensuring that operations can run safely in parallel.

Parallel Collections in Scala:

- ParSeq, ParMap, etc. parallelize operations like map, filter, and reduce.
- Use .par

Distributed Collections:

- In distributed programming, collections are processed across multiple machines (e.g., Apache Spark's RDDs).
- Functions like map, flatMap, and reduce are distributed and executed on different partitions of the dataset.

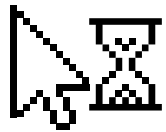
Exercise

Design a Scala program that processes a large dataset of employees and their work details. The dataset consists of employee records, represented as:

```
case class Employee(id: Int, name: String, department: String, salary: Double,
projects: Set[String])
```

Given this list of employees:

```
val employees = List(
    Employee(1, "Alice", "Engineering", 85000.0, Set("ProjectA", "ProjectB")),
    Employee(2, "Bob", "Engineering", 90000.0, Set("ProjectC")),
    Employee(3, "Charlie", "HR", 60000.0, Set("ProjectD", "ProjectE",
"ProjectF")),
    Employee(4, "Diana", "Engineering", 120000.0, Set("ProjectA")),
    Employee(5, "Eve", "HR", 65000.0, Set("ProjectE", "ProjectF")),
    Employee(6, "Frank", "Marketing", 70000.0, Set("ProjectG"))
)
```



01 Task 1: Filter Employees by Department and Salary:

```
val highEarningEngineers = employees.filter(emp ⇒ emp.department == "Engineering" &&
emp.salary > 80000)
println(highEarningEngineers)
// Output: List(Employee(1,Alice,Engineering,85000.0,Set(ProjectA, ProjectB)), ...
```

02 Task 2: Map to Names and Project Counts:

```
val engineerProjectCounts = highEarningEngineers.map(emp ⇒ (emp.name,
emp.projects.size))
println(engineerProjectCounts)
// Output: List((Alice,2), (Bob,1), (Diana,1))
```

03 Task 3: Generate a Flat List of All Unique Project Names:

```
val allProjects = employees.flatMap(_.projects).toSet
println(allProjects)
// Output: Set(ProjectA, ProjectB, ProjectC, ProjectD, ProjectE, ProjectF, ProjectG)
```

04 Task 4: Calculate Total Salary Using FoldLeft

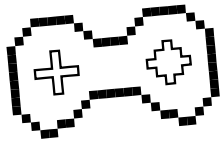
```
val totalSalary = employees.foldLeft(0.0)((acc, emp) ⇒ acc + emp.salary)
println(totalSalary)
// Output: 490000.0
```

05 Task 5: Find the Employee with the Highest Salary Using Reduce

```
val highestPaidEmployee = employees.reduce((emp1, emp2) ⇒ if (emp1.salary >
emp2.salary) emp1 else emp2)
println(highestPaidEmployee)
// Output: Employee(4,Diana,Engineering,120000.0,Set(ProjectA))
```

05 Task 6: Compute Total Salary Using Parallel Collections

```
val parallelTotalSalary = employees.par.map(_.salary).reduce(_ + _)
println(parallelTotalSalary)
// Output: 490000.0
```



Thanks!

Valeria Saldana

Source:https://csci3155.cs.colorado.edu/pppl-course/book/higher-order-functions.html#sec-abstract-data-types-and-higher-order-functions