06 Lab
Advanced mechanisms of the Scala language

Mirko Viroli, Roberto Casadei, Gianluca Aguzzi
{mirko.viroli,roby.casadei,gianluca.aguzzi}@unibo.it

C.D.L. Magistrale in Ingegneria e Scienze Informatiche
Alma Mater Studiorum—Università di Bologna, Cesena

a.a. 2022/2023

Lab 06: Outline

Outline

- Consolidate your knowledge of Scala
- Practice with advanced Scala features

Repo with exercises

- Fork/clone `https://github.com/unibo-pps/pps-22-23-lab06`
- Open the provided Scala project in IntelliJ
- The code for this lab is in package **u06lab.code**
- You may want to copy such code in a package `u06lab.solution` so that the problems and the corresponding solutions are kept separate
- For each exercise, you are given a (statically correct) code template that you have to complete as well as a main program to be executed for checking your solution and making experiments
- As usual, you may commit your changes and push them to your own (forked) repository

Exercise 1: `Combiner`

1) Implement `FunctionsImpl` such that the code in `TryFunctions` works correctly.
   - **N.B.**: complete this before looking at the following step!
2) To apply DRY principle at the best, note the three methods in Functions do something similar. Use the following approach (called type classes approach).

- find three implementations of `Combiner` that tell (for `sum`, `concat` and `max`) how to combine two elements, and what to return when the input list is empty.
  - Observe how much they are both structurally and functionally similar.
  - `Combiner[T]` (typically call `Monoid` in FP context) is called a *type class* since it is a mechanism to conceptually add operations to type `T`
- Implement in `FunctionsImpl` a **new method** `combine` that, other than the collection of `A`s, takes a `Combiner` object as parameter too
- Implement the three methods by simply calling `combine`
- When all works, note we completely avoided duplications.

3) Note that `combine` could take the `Combiner` with `using` clause

Exercise 2: `Parser`

1) Provide missing implementations such that the code in `TryParsers` works correctly.
   - Consider the `Parser` example shown in previous lesson.
   - Analogously to `NonEmpty`, create a mixin `NotTwoConsecutive`, which adds the idea that one cannot parse two consecutive elements which are equal.
   - Use it (as a mixin) to build class `NotTwoConsecutiveParser`, used in the testing code at the end.
   - Note we also test that the two mixins can work together!!
   - Write the full linearisation of `parserNTCNE`
   - **N.B.:** tests are written in such a way that each call to `parseAll` runs on a brand-new parser (got via a 0-arg `def`). If you want to avoid this (i.e., running `parseAll` multiple times on the same parser object), you need to reset the parser after use (e.g., in `parseAll`)
2) Extend Scala type `String` with a factory method that creates a parser which recognises the set of chars of a string.
3) **Optional** Implment mixin `ShortenThenN` which accepts a sequence of chars of length at most `n` (part of the trait constructor).

Exercise 3: Solitaire game

Consider the solitaire game described here:
`http://www.luigilamberti.it/Software/Sol35/Sol35.htm`

- **Description**: A board with dimensions `width`(w) x `height`(h) is given (start with 5x5), and the objective is to place a total of $w * h$ numbers (25 in this case) on the board according to specific rules:
  1. The game starts with the player placing a number in the center of the board, at position `(width/2, height/2)`.
  2. From there, the player can move the number to any adjacent position, either vertically or horizontally by two positions, or diagonally by one position
     - e.g., you start from (2, 2), you can move to (0, 2), (1, 1), (1, 3), (2, 0), (2, 4), (3, 1), (3, 3), (4, 2)
  3. The player must continue placing numbers on the board until all positions are filled, making sure that the number being placed is not already occupied
- **Goal**: implement a function `placeMarks` that, given a board (w x h), computes all the possible solutions
- **Hints**:
  - A solution can be represented as a list of positions
  - Follow the structure of the eight queens problem demonstrated in class

Exercise 4 (Optional): `ConnectThree`

Follow the exercises sketched in object `ConnectThree` – a simplified version of `ConnectFour` in which the board is 4x4 and a player wins with three aligned disks

1. Implement `find` such that the code provided behaves as suggested by the comments
2. Implement `placeAnyDisk` such that the code provided behaves as suggested by the comments
3. Implement `firstAvailableRow` following the output provided in the comments
4. (Advanced) Implement `computeAnyGame` such that the code provided behaves as suggested by the comments
5. (Very advanced) Modify the above one so as to stop each game when someone won
6. (Optional) Try to implement TicTacToe logic following this structure