Scala

how to use it in production

Good practices

Your code should be

- Safe (immutability, typing, testing, versioning)

- Maintainable (organized, good patterns, guidelines/linter)

- Simple (do not anticipate too much, do not complexify for the sake of it)

- Performant (when necessary and with the right balance, don't forget the previous points)

Useful Scala types - Option

```scala
val javaStuff = null
val safeJavaValue = Option(javaStuff)

// NPE at runtime
val result = javaStuff * 2
// Can't fail
val safeResult = safeJavaValue.map(v => v*2).getOrElse(0)
```

- A value may or may not be present, which happens a lot in a real codebase
- Working with options forces you to think about it
- Easy to manipulate and compose (map, flatMap, …)

Useful Scala types - Try

```scala
val process = Try(dangerousStuff)

process match {
  case Success(v) => v * 2
  case Failure(ex) =>
    println(s"An error occured $ex, fallbacking on default value")
    0
}
```

- Manage and catch exceptions
- Represent the fact that it can fail in the type
- Easy to manipulate and compose (map, flatMap, …)

Useful Scala types - Either

```scala
val getAge: Either[Exception, Int] = Right(28)
val getName: Either[Exception, String] = Left(Exception.EmptyUser)

getAge.flatMap(age => getName.map(name => "My name is $name and I am $age years old"))
```

- Useful for manipulating business errors
- Right biased (so correct value on the right, error on the left)
- Easy to manipulate and compose (map, flatMap, …)

For comprehension

```scala
val getAge: Either[Exception, Int] = Right(28)
val getName: Either[Exception, String] = Left(Exception.EmptyUser)
val getPlaceOfBirth: Either[Exception, String] = Right("Béziers")

val getPerson: Either[Exception, Person] = for {
  age <- getAge
  name <- getName
  placeOfBirth <- getPlaceOfBirth
} yield Person(age, name, placeOfBirth)
```

- Syntactic sugar for `.map`, `.flatMap` (and `withFilter`) combinations

Value classes

```scala
final case class Age(val underlying: Int) extends AnyVal {
  def isAdult: Boolean = underlying >= 18
}
```

- No memory overhead at runtime
- Avoid mixing values in your code
- https://github.com/fthomas/refined to go even further with refinement types

Manage effects

```scala
def headsOrTails(tossCoin: => Boolean): Side = {
  if (tossCoin) Side.Head else Side.Tail
}

// production
headsOrTails(Random.nextBoolean)

// test
assert(headsOrTails(true) == Side.Head)
```

- Your effects should be represented by a specific type
- Make sure you can mock them easily in your tests

Future and IO

- Represent an asynchronous call
  You don't need to manage it yourself

- Compose them
  And avoid any blocking in your code

- Many implementations

  Vanilla `Future` https://docs.scala-lang.org/overviews/core/futures.html
  Cats-effect `IO` https://typelevel.org/cats-effect/datatypes/io.html
  Monix `Task` https://monix.io/docs/current/eval/task.html
  ZIO `ZIO` https://zio.dev/docs/overview/overview_index

```scala
implicit val executionContext = ...
val call: Future[Int] = Future { ... }

call.onComplete
call.map
call.flatMap(v => call2(v))
```

Future and IO

- ## At some point, you need to await
  Do it once and for all in your main

- ## `Future` is **eager**

  In comparison to being **lazy** like `IO` and `Task`

```scala
val getName = Future { ... }
val getId = Future { ... }

val result = for {
  name <- getName
  id <- getId
  age <- getAge(id, name)
} yield age

// Blocking, only run once at the end

import scala.concurrent.duration._
Await.result(result, 1 minute)
```
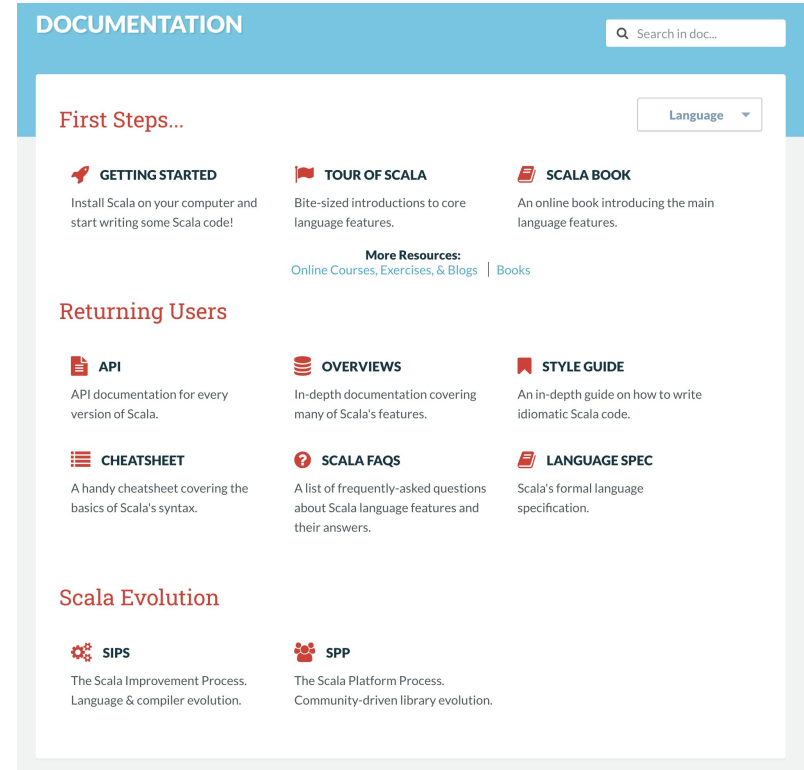
Keep the documentation nearby

- Getting started and overview
  https://docs.scala-lang.org/

- Well maintained

  And a good place to find next evolutions

Questions ?

Exercises - Write a blackjack game

Rules

- The goal of blackjack is to beat the dealer's hand without going over 21
- Face cards are worth 10. Aces are worth 1 or 11, whichever makes a better hand
- Each player starts with two cards, one of the dealer's cards is hidden until the end
- To 'Hit' is to ask for another card. To 'Stand' is to hold your total and end your turn
- If you go over 21 you bust, and the dealer wins regardless of the dealer's hand
- If you are dealt 21 from the start (Ace & 10), you got a blackjack
- Blackjack means you win 2x the amount of your bet, else you win or lose your bet depending on the result
- Dealer will hit until his/her cards total 17 or higher
- The player hits first
- The player bets credits for the round before the hands are distributed
- You can stop after any round, but you have to stop if your credits reach 0

Exercise 1 - Bootstrap

Let's go !

- Create a new sbt project with latest sbt and scala versions called `blackjack-polytech`

- Create an arborescence with a package `fr.polytech.blackjack` and a Main printing what the game is about

Exercise 2 - Models

A blackjack game is played with french cards, so let's represent them

- Create a `Rank` enumeration with all the different cards and their values

- Create a `Suit` enumeration with the 4 different colors

- Create a `Card` model which is the combination of a `Rank` and a `Suit`

- Generate a deck in the `Main` class

Exercise 3 - Manage a hand

Now let's represent a hand - while keeping everything immutable

- Create a `Hand` model which contains a list of `Card`

- Implement the different calculating methods (value, specialValue - as an Ace can be either a 1 or a 11, isBlackjack, isBust)

- Implement a way to add a card to the `Hand`, and show cards based on the user or dealer view (user show all cards, dealer may only show the first)

- Implement a hand comparison to find the winner between two of them

Exercise 4 - Start game and draw hands

- Create a `Blackjack` class responsible for running the game

- Start the game by defining the initialCredits, specifying the bet for the round, and then draw the hand for both players

- Create a method to show both hands, but keep in mind the dealer only shows the first card of his hand until the end

Exercise 5 - Run the game

- Need methods to check the current and final results

- Manage player steps (hit or stand based on their decision)

- Manage dealer steps (hit until the value of their hand is above 17)

- Don't forget that a player wins or loses if they hit blackjack or bust, without having to play the dealer step

- Manage the rounds and the credits update based on a round result (the game stops at the end of a round if the player wants to leave or doesn't have any more credits)

Exercise 6 - Add safety

- Use value classes to differentiate player and dealer hands

- Add tests and update your implementation to manage effects more clearly

- What about representing all println by a proper effect to test them and change the result later ?

Repository

https://github.com/Elyrixia/blackjack-example
With a commit for each step