Part #1 - Introduction

1. Title, logo, introduce yourself
2. Agenda: Introduce main FP concepts, implement examples with Scala, showcase Scala plugin
3. Why FP? Discuss how FP concepts help each other
4. Why Scala? Discuss how the user can start with imperative code and move towards FP gradually, what is JVM, and how other programming languages compare to Scala
5. Why Scala Plugin? Discuss what is an IDE, what is the position of IntelliJ IDEA on the market, how Scala Plugin supports Scala.
6. Learning materials: Talk about the Academy plugin, the interactive course, IntelliJ Scala Bundle + onboarding tips, and the most popular places in the internet to learn Scala (also the main Scala webpage)

Part #2 - Functions as Data

Showcase: type inference, "Can Be Private" inspection, "Unused Element" inspection, enums import, method names suggestion (for collections), "remove expression" action for unused elements, match/case code completion (in fizzBuzz). Inlay hints and X-Ray.

Ctrl + Shift + Enter -> Complete current statement

Option + Shift + Space -> Type-Matching

Option + Shift + Backspace -> Line completion

1. What is a function
2. Anonymous functions
3. Passing functions as arguments
4. Case class

Functions as data (1)

```
object FunctionsAsData {
   class User(val name: String, val email: String) {
    override def toString: String = s"$name:$email"
  }
```

```scala
  // Let's create a sequence of three users
  val users: Seq[User] = Seq(
    User("Joe", "joe@gmail.com"),
    User("Felix", "felix@proton.me"),
    User("Garfield", "garfield@catmail.org")
  )

  // A function which checks if the user's email is valid
  def isEmailValid(user: User): Boolean = user.email.contains('@')

  // Pass the appropriate function into `filter` to create a sequence of
users with valid emails.
  private val usersWithValidEmails = users.filter(isEmailValid)

  // The whole "User => Boolean" is the type of the function
  private val isEmailValidFun: User => Boolean =
    user => user.email.contains('@')

  // We can use it instead of the method version in `filter`
  val usersWithValidEmailsFun: Seq[User] = users.filter(isEmailValidFun)

  // or we can use an anonymous function - also called lambdas
  val usersWithValidEmailsAnon: Seq[User] = users.filter(user =>
user.email.contains('@'))

  // Lambdas, just as all functions, can take a value from its scope
  private val alwaysValidName = "Maciek"
  val usersWithValidEmailsOrMaciek: Seq[User] =
    users.filter(
      user => user.name == alwaysValidName || isEmailValid(user)
    )

/*  @main def main(): Unit = {
    println(usersWithValidEmails)
  }*/
}
```

Scala collections: filter, find, foreach, map (2)

```
object Collections {
  case class User(name: String, email: String)

  val joe = User("Joe", "joe@gmail.com")
  val felix = User("Felix", "felix@proton.me")
  val garfield = User("Garfield", "garfield@catmail.org")

  // Scala collections: Seq, Set, Map
  val usersSeq = Seq(joe, garfield, felix)
  val sortedUsers = usersSeq.sortWith( (u1, u2) => u1.name < u2.name )

  // This won't work because users don't have an implicit ordering
  //val sortedUsers2 = usersSeq.sorted

  // But this will
  val sortedUsers3 = usersSeq.sortBy(_.name)

  // maps
  val usersMap = Map(
    "Felix" -> felix,
    "Garfield" -> garfield,
    "Joe" -> joe
  )

  // a map can be thought of as a set of tuples
  val usersMap2 = usersSeq.map(u => u.name -> u).toMap

  // tuples
  val userTuples: Seq[(String, User)] = usersMap.toSeq

  // filter - we already discussed it

  // find
  val protonEmail: Option[User] = usersSeq.find { user =>
user.email.endsWith("@proton.me") }
  val felixTheUser: Option[User] = usersSeq.find { joe => joe.name ==
"Felix" } // also: _.name == "Felix"

  // foreach
  usersSeq.foreach(user => println(s"This is ${user.name}"))

  // map
```

```scala
  // Let's create a conversation class
  case class Conversation(title: String, participants: Seq[User])
  // And let's create conversations of Joe and every other user
  val joesConvos: Seq[Conversation] =
    usersSeq
      .filterNot(_.name == "Joe")
      .map(user => Conversation(s"Joe and ${user.name}", Seq(joe,
user)))
}
```

Collections FlatMap (3)

```scala
object CollectionsFlatMap {
  // Let's take that map for Joe and use it for every user
  case class User(name: String, email: String)
  case class Conversation(title: String, participants: Seq[User])

  val joe: User = User("Joe", "joe@gmail.com")
  val felix: User = User("Felix", "felix@proton.me")
  val garfield: User = User("Garfield", "garfield@catmail.org")
  val users: Seq[User] = Seq(joe, garfield, felix)

  val convos: Seq[Conversation] =
    users.flatMap { user =>
      users
        .filterNot(_.name == user.name)
        .map(user2 => Conversation(s"${user.name} and ${user2.name}",
Seq(user, user2)))
    }
/*
  @main def main(): Unit = {
    println(s"Number of convos: ${convos.size}")
    convos.foreach(c => println(c.title))
  }*/
}
```

Collections FoldLeft (4)

```
  // fizz buzz! union types!
  def fizzBuzz(n: Int): Int | String = n match {
    case _ if n % 15 == 0 => "FizzBuzz"
    case _ if n % 3 == 0  => "Fizz"
    case _ if n % 5 == 0  => "Buzz"
    case _ => n
  }

  // Generate a range of numbers from 1 to 100
  val numbers = 1 to 100

  // Use foldLeft to iterate through the numbers and apply the fizzBuzz
function
  val fizzBuzzList =
    numbers.foldLeft[List[Int | String]](Nil) {
      (acc, n) => acc.appended(fizzBuzz(n))
    }

/*  @main def main(): Unit = {
    println(fizzBuzzList)
  }*/
```

Collections X-Ray (5)

Scala supports **implicit conversions** and **implicit parameters**, which can significantly reduce boilerplate code - but it might be challenging to understand where values come from or how types are converted. Implicit hints show information about implicit arguments and implicit conversions that the data in question will undergo during compilation. The X-Ray mode may prove to be particularly useful in this case, letting you see implicit hints only for a moment when you want to make sure what they are, and then they will disappear again.

Partial Functions (6)

```
object PartialFunctions {
```

```
    // Using the collect method, create a set of names of parents of our
  users

    import org.fpinscala.UserData.parents
    import org.fpinscala.UserData.database

    val parentNames: Seq[String] =
      database
        .map(u => parents.get(u.id))
        .collect {
          case Some(id) if database.exists(_.id == id) =>
  database(id).name
        }

    import org.fpinscala.Animal.animals
    import org.fpinscala.Cat
    import org.fpinscala.Color.Black

    // Using the collect method, create a sequence of animals containing
  only black cats
    val blackCats: Seq[Cat] = animals.collect {
      case cat: Cat if cat.color == Black => cat
    }

    // We will come back to `collect` in the part about early returns
  }
```

Functions returning functions (7)

Currying is the process of converting a function with multiple arguments into a sequence
of functions that take one argument. Each function returns another function that consumes
the following argument.

Partial application is the process of reducing the number of arguments by applying some
of them when the method or function is created.

More: https://www.baeldung.com/scala/currying

```
  import scala.util.Random
```

```scala
object FunctionsReturningFunctions {
  case class Cat(name: String, color: Color)

  val colors = Color.values.toSeq
  val catNames = Seq("Fluff", "Shadow", "Garfield")

  def catGenerator(name: String): () => Cat = {
    val color = colors(Random.nextInt(colors.size))
    () => Cat(name, color)
  }

  def catGenerator(color: Color): () => Cat = {
    val name = catNames(Random.nextInt(catNames.size))
    () => Cat(name, color)
  }

  val gen1 = catGenerator("Fluff")
  val gen2 = catGenerator(Black)

  // partial application
  def createBlackCat = Cat.apply(_, Color.Black)

  def sum(x: Int, y: Int): Int = x + y
  def sum2(x: Int)(y: Int): Int = x + y
  val sum3: Int => Int => Int = { x => y => x + y }
  val increment: Int => Int = sum2(1)(_)

/*  @main def main(): Unit = {
    //println(gen1())
    //println(gen2())
    //println(increment(3))
  }*/
}
```

Early returns (8)

```scala
  import UserData.database

  /**
   * This is our "complex conversion" method.
```

```scala
   * We assume that it is costly to retrieve user data, so we want to
avoid
   * calling it unless it's absolutely necessary.
   *
   * This version of the method assumes that the user data always exists
for a given user id.
   */
  def complexConversion(userId: UserId): UserData =
    database.find(_.id == userId).get

  /**
   * Similar to `complexConversion`, the validation of user data is
costly
   * and we shouldn't do it too often.
   */
  def complexValidation(user: UserData): Boolean =
    !user.email.contains(' ') && user.email.count(_ == '@') == 1

  private val userIds: Seq[UserId] = 1 to 11

/*  @main def main(): Unit = {
    val user = Imperative.findFirstValidUser(userIds)
    println(user)
  }*/

  /**
   * Imperative approach that uses unidiomatic `return`.
   */
  object Imperative {
    def findFirstValidUser(userIds: Seq[UserId]): Option[UserData] = {
      for (userId <- userIds) {
        val userData = complexConversion(userId)
        if (complexValidation(userData)) return Some(userData)
      }
      None
    }
  }

  /**
   * Naive functional approach: calls `complexConversion` twice on the
selected ID.
   */
  object Naive {
    def findFirstValidUser(userIds: Seq[UserId]): Option[UserData] =
```

```
      userIds
        .find(userId => complexValidation(complexConversion(userId)))
        .map(complexConversion)
  }

  /**
   * A more concise implementation, which uses `collectFirst`.
   */
  object CollectFirst {
    def findFirstValidUser(userIds: Seq[UserId]): Option[UserData] =
      userIds.collectFirst {
        case userId if complexValidation(complexConversion(userId)) =>
complexConversion(userId)
      }
  }

  /**
   * The custom `unapply` method runs conversion and validation and only
returns valid user data.
   */
  object CustomUnapply {
    object ValidUser {
      def unapply(userId: UserId): Option[UserData] =
        val userData = complexConversion(userId)
        if complexValidation(userData) then Some(userData) else None
    }

    def findFirstValidUser(userIds: Seq[UserId]): Option[UserData] =
      userIds.collectFirst {
        case ValidUser(user) => user
      }
  }

  /**
   * This function takes into account that some IDs can be left out from
the database
   */
  def safeComplexConversion(userId: UserId): Option[UserData] =
database.find(_.id == userId)

  /**
   * Partiality of `safeComplexConversion` trickles into the search
function.
   */
```

```scala
  object SafeImperative {
    def findFirstValidUser(userIds: Seq[UserId]): Option[UserData] =
      for (userId <- userIds) {
        safeComplexConversion(userId) match {
          case Some(user) if complexValidation(user) => return
Some(user)
          case _ =>
        }
      }
      None
  }

  /**
   * This custom `unapply` method performs the safe conversion and then
validation.
   */
  object SafeCollectFirst {
    object ValidUser {
      def unapply(userId: UserId): Option[UserData] =
        safeComplexConversion(userId).find(complexValidation)
    }

    def findFirstValidUser(userIds: Seq[UserId]): Option[UserData] =
      userIds.collectFirst {
        case ValidUser(user) => user
      }
  }

  object MoreThanOneValidation {
    object ValidUser {
      def unapply(userId: UserId): Option[UserData] =
        safeComplexConversion(userId).find(complexValidation)
    }

    object ValidUserInADifferentWay {
      def otherValidation(userData: UserData): Boolean = false
      def unapply(userId: UserId): Option[UserData] =
safeComplexConversion(userId).find(otherValidation)
    }

    def findFirstValidUser(userIds: Seq[UserId]): Option[UserData] =
      userIds.collectFirst {
        case ValidUser(user) => user
        case ValidUserInADifferentWay(user) => user
```

```scala
      }
  }

  object Deconstruct {
    trait Deconstruct[From, To] {
      def convert(from: From): Option[To]
      def validate(to: To): Boolean
      def unapply(from: From): Option[To] = convert(from).find(validate)
    }

    object ValidUser extends Deconstruct[UserId, UserData] {
      def convert(userId: UserId): Option[UserData] =
safeComplexConversion(userId)
      def validate(user: UserData): Boolean = complexValidation(user)
    }

    def findFirstValidUser(userIds: Seq[UserId]): Option[UserData] =
      userIds.collectFirst {
        case ValidUser(user) => user
      }
  }

  object LazyCollection {
    def findFirstValidUser(userIds: Seq[UserId]): Option[UserData] =
      userIds
        .iterator
        .map(safeComplexConversion)
        .find(_.exists(complexValidation))
        .flatten
  }

  import scala.util.boundary
  import scala.util.boundary.break

  object BreakingBoundary {
    def findFirstValidUser(userIds: Seq[UserId]): Option[UserData] = {
      boundary:
        for (userId <- userIds)
          safeComplexConversion(userId).foreach { userData =>
            if (complexValidation(userData)) break(Some(userData))
          }
        None
    }
  }
```

Lazy collections (9)

```
object CollectionView {
  private val numbers: Seq[Int] = 1 to 100

  // Without using view
  val firstEvenSquareGreaterThan100_NoView: Int =
    numbers
      .map(n => n * n)
      .filter(n => n > 100 && n % 2 == 0)
      .head

  // Using view
  val firstEvenSquareGreaterThan100_View: Int =
    numbers
      .view
      .map(n => {
        println(s"Square of $n being calculated.") // To observe the
lazy evaluation
        n * n
      })
      .filter(n => n > 100 && n % 2 == 0)
      .head

  @main def main(): Unit = {
    println(firstEvenSquareGreaterThan100_NoView)
    println(firstEvenSquareGreaterThan100_View)
  }
}
```

Part #3 - The type system and recursion

Showcase the debugger, enum suggestions, enum imports, match/case exhaustive, the recursion icon, "the recursive call requires the return type", "cannot rewrite recursive call in tail position".

Recursion (10)

```scala
import scala.annotation.tailrec
import scala.util.control.TailCalls.*

object Recursion {
  def factorial(n: Int): BigInt =
    if (n <= 0) 1
    else n * factorial(n - 1)

  /*
  ```scala
  factorial(3)
  3 * factorial(2)
  3 * 2 * factorial(1)
  3 * 2 * 1 * factorial(0)
  3 * 2 * 1 * 1
  3 * 2 * 1
  3 * 2
  6
  ```
   */

  def tailFactorial(n: Int): BigInt = {
    @tailrec
    def go(n: Int, accumulator: BigInt): BigInt =
      if (n <= 0) accumulator
      else go(n - 1, n * accumulator)

    go(n, 1)
  }

  def trampolineFactorial(n: Int): BigInt = {
    def go(i: Int): TailRec[BigInt] =
      if (i <= 0) done(1)
      else tailcall(go(i - 1)).map(_ * i)

    go(n).result
  }

  // 1 -> 1, 2 -> 1, 3 -> 2, 4 -> 3, 5 -> 5, 6 -> 8

  def fibonacci(n: Int): BigInt =
    if (n == 1) 1
```

```scala
    else if (n == 2) 1
    else fibonacci(n - 2) + fibonacci(n - 1)

  def tailFibonacci(n: Int): BigInt = {
    @tailrec
    def go(currentN: Int, currentSum: BigInt, previousSum: BigInt):
BigInt = {
      if (currentN < 1) 1
      else if (currentN == 1) previousSum
      else go(currentN - 1, currentSum + previousSum, currentSum)
    }

    go(n, 1, 1)
  }

  def trampolineFibonacci(n: Int): BigInt = {
    def go(i: Int): TailRec[BigInt] = i match {
      case 1 => done(1)
      case 2 => done(1)
      case _ =>
        for {
          a <- tailcall(go(i - 1))
          b <- tailcall(go(i - 2))
        } yield a + b
    }

    go(n).result
  }

  private val fib: LazyList[BigInt] =
    BigInt(1) #::
      BigInt(1) #::
      fib.zip(fib.tail).map { case (a, b) => a + b }

  def fibonacciLazyList(n: Int): BigInt = fib.take(n).last

  @main def main(): Unit = {
    //val res = factorial(10000)
    //val res = tailFactorial(10000)
    //val res = trampolineFactorial(10000)
    //val res = fibonacci(10)
    //val res = tailFibonacci(6)
    val res = trampolineFibonacci(6)
    //val res = fibonacciLazyList(6)
```

```
    println(res)
  }
}
```

ADT, enums, trait hierarchies (11)

```
object Enums {
  enum Tree[+T] { // The + in +T is here because of covariance.
    case Branch(left: Tree[T], value: T, right: Tree[T])
    case Leaf(value: T)
    case Stump
  }

  import Tree.*

  /*
      3
     / \
    2   5
   /   / \
  1   4   6
  */

  val intTree: Tree[Int] =
    Branch(
      Branch(Leaf(1), 2, Stump),
      3,
      Branch(Leaf(4), 5, Leaf(6))
    )

  def print[T](tree: Tree[T], toString: T => String, level: Int = 0):
Unit = {
    val prefix = Seq.fill(level)('-').mkString
    tree match { // show match exhaustive!
      case Branch(left, value, right) =>
        println(s"$prefix${toString(value)}")
        print(left, toString, level + 1)
        print(right, toString, level + 1)
      case Leaf(value) =>
        println(s"$prefix${toString(value)}")
      case Stump =>
        println(s"${prefix}X")
```

```
    }
  }

  def add[T](tree: Tree[T], t: T, compare: (T, T) => Int): Tree[T] =
tree match {
    case Branch(left, value, right) if compare(t, value) < 0 =>
      val newLeft = add(left, t, compare)
      Branch(newLeft, value, right)
    case Branch(left, value, right) if compare(t, value) > 0 =>
      val newRight = add(right, t, compare)
      Branch(left, value, newRight)
    case branch@Branch(_, _, _) =>
      branch // no changes
    case Leaf(value) if compare(t, value) < 0 =>
      Branch(Leaf(t), value, Stump)
    case Leaf(value) if compare(t, value) > 0 =>
      Branch(Stump, value, Leaf(t))
    case leaf@Leaf(_) =>
      leaf // no changes
    case Tree.Stump =>
      Leaf(t)
  }

  def add[T](tree: Tree[T], values: Seq[T], compare: (T, T) => Int):
Tree[T] =
    values.foldLeft[Tree[T]](tree) { (tree, t) => add(tree, t, compare)
}

  def create[T](values: Seq[T], compare: (T, T) => Int): Tree[T] =
add(Stump, values, compare)

  /*@main*/ def main(): Unit = {
    //print(intTree, _.toString)
    val compare = (a: Int, b: Int) => a - b
    val newTree: Tree[Int] =
      Seq(10, 5, 15, 2, 8)
        .foldLeft[Tree[Int]](Stump) { (tree, t) => add(tree, t, compare)
}
     // val newTree = create(Seq(10, 5, 15, 2, 8), compare)
    print(newTree, (n: Int) => n.toString)
  }

}
```

Tree traits

```
sealed trait Tree[+T]

object Tree {
  case class Branch[T](left: Tree[T], value: T, right: Tree[T]) extends
Tree[T]
  case class Leaf[T](value: T) extends Tree[T]
  case object Stump extends Tree[Nothing]
}
```

Monads (12)

```
  import UserData.{database, parents}

  object WithExceptions {
    def findParent(name: String): UserData = {
      val user = UserService.findUser(name) // Java
      val parentId = parents(user.id)
      database.find(_.id == parentId).get
    }
  }

  object WithOption {
    def findParent(name: String): Option[UserData] =
      database
        .find(_.name == name)
        .flatMap(user => parents.get(user.id))
        .flatMap(parentId => database.find(_.id == parentId))

    def findParent2(name: String): Option[UserData] =
      for {
        user     <- database.find(_.name == name)
        parentId <- parents.get(user.id)
        parent   <- database.find(_.id == parentId)
      } yield parent
  }
```

```scala
  object WithEither {
    def findParent(name: String): Either[String, UserData] =
      database
        .find(_.name == name).toRight(s"No user $name in the database")
        .flatMap(user => parents.get(user.id).toRight(s"No parent found
for user $name"))
        .flatMap(parentId => database.find(_.id ==
parentId).toRight(s"No parent with id $parentId found in the database"))

    def findParent2(name: String): Either[String, UserData] =
      for {
        user     <- database.find(_.name == name).toRight(s"No user
$name in the database")
        parentId <- parents.get(user.id).toRight(s"No parent found for
user $name")
        parent   <- database.find(_.id == parentId).toRight(s"No parent
with id $parentId found in the database")
      } yield parent
  }

  @main def main(): Unit = {
    val name = read()
    println(s"name: $name")

    // exception
    try {
      val user = WithExceptions.findParent(name)
      println(user)
    } catch {
      case ex: Throwable => println(ex.getMessage)
    }

    // Option
    WithOption.findParent(name) match {
      case Some(parent) => println(parent)
      case None         => println("Error")
    }

    // Either
    WithEither.findParent(name) match {
      case Left(error) => println(s"Error: $error")
      case Right(parent) => println(parent)
    }
```

```
    // Try
    Try(WithExceptions.findParent(name)) match {
      case Failure(exception) => println(s"Error:
${exception.getMessage}")
      case Success(parent)    => println(parent)
    }
  }

  // try out with:
  // John Doe (-> Michael Brown), Maciek (-> not in the database), Jane
Smith (-> parent not in the database),
  // Emily Johnson (-> invalid parent id)

  private def read(): String = {
    printf("\n> ")
    readLine().trim
  }
```

Cats Effect (13)

```
package org.fpinscala.finished

import cats.effect.{IO, IOApp}
import cats.syntax.all.*
import org.fpinscala.UserData

import java.nio.file.{Files, Path}
import scala.jdk.CollectionConverters.*

object CatsEffectVersion extends IOApp.Simple {
  override def run: IO[Unit] =
    for {
      lines    <- read(UserData.FilePath)
      users    =  lines.map(UserData.fromLine)
      n        <- askForUpdate
      updated  <- users.traverse(updateAge(_, n))
      newLines =  updated.map(_.toLine)
      _        <- write(UserData.FilePath, newLines)
    } yield ()
```

```
  private def read(path: Path): IO[List[String]] =
    IO { Files.readAllLines(path).asScala.toList }

  private val askForUpdate: IO[Int] =
    for {
      _      <- IO.print("By how much should I update the age? ")
      answer <- IO(scala.io.StdIn.readLine())
    } yield answer.toInt

  private def updateAge(user: UserData, n: Int): IO[UserData] = {
    val newAge = user.age + n
    for {
      _       <- IO.println(s"The age of ${user.name} changes from
${user.age} to $newAge")
      updated =  user.copy(age = newAge)
    } yield updated
  }

  private def write(path: Path, lines: List[String]): IO[Unit] =
    IO { Files.writeString(path, lines.mkString("\n")) }
}
```