- = Introduction to Functional Programming with Scala
- ==1. Title Slide Introduction to Functional Programming with Scala
  - Duration: 30 hours
  - Prerequisites: Basic programming knowledge, High school mathematics

- == 2. Teacher Profile
  - Instructor: Said Boudjelda
  - ► Background:
    - ► Senior Software Engineer @Sciam
    - ▶ Open source contributor
  - Contact:
    - Email: mohamed-said.boudjelda@intervenants.efrei.net
    - Office Hours: 30
  - ► Teaching Philosophy: Empowering students through mathematical rigor and practical coding skills
- [NOTE] Reach out for guidance or project ideas!

== 3. Course Description This introductory course explores functional programming principles using Scala. Through synergy of mathematical concepts and hands-on coding, you'll learn how to design reliable, maintainable software in the functional paradigm.

#### Key Focus:

- Bridging mathematical logic to coding practice
- Applying set theory, algebra, and logic in programming
- ▶ Writing robust, immutable Scala programs

- == 4. Learning Objectives
  - Understand mathematical foundations of functional programming
  - Grasp basic algebraic structures relevant to programming
     Write basic Scala programs using functional programming concepts
  - Understand and use immutable data structures
  - Work with Scala collections and their operations
  - and more

- == 5. Course Structure and Topics
- Week 1: Mathematical Foundations and Introduction to Scala I (4 hours)
- Week 2: Introduction to Scala II (3 hours)

...

Focus today: Week 1 (Mathematics and Scala Basics)

- == 6. Assessment Overview Course Assessments:
- Assignments (20%): Weekly coding tasks and mathematical exercises
  - Focus:
  - Due.
- ▶ Midterm Project (20%): Design a functional Scala application
  - Example:
  - Due.
- Final Exam (50%): Written and coding components
  - Topics:
    - Date:
- Participation (10%): In-class discussions, group activities

[TIP] Start assignments early and use office hours!

- == 7. References and Recommended Reading *Books*:
  - ▶ Paul Chiusano and Runar Bjarnason, Functional Programming in Scala (Manning, 2014)
- Martin Odersky, Lex Spoon, and Bill Venners, *Programming in Scala* (Artima, 5th Ed., 2021)
- ➤ Kenneth Rosen, *Discrete Mathematics and Its Applications* (McGraw-Hill, 8th Ed., 2018)

#### Online Resources:

- ▶ link:https://docs.scala-lang.org/[Scala Documentation]
- ▶ link:https://www.scala-exercises.org/[Scala Exercises]
- ▶ link:https://www.discrete-math.org/[Discrete Math for Programmers]

## Papers:

- ▶ Philip Wadler, "Monads for Functional Programming" (1992)
- Martin Odersky et al., "An Overview of the Scala Programming Language" (2006)

[TIP] Use these to deepen your theoretical and practical knowledge!

- == 8. Course Policies Attendance:
  - Expected; notify instructor for absences
- Missed classes: Review slides, consult peers/instructor

## Academic Integrity:

- ► Work must be original
- Plagiarism or collaboration without permission: Failing grade for assignment, potential course failure
- Cite external code/resources

[NOTE] Clear communication ensures a smooth experience!

- == 9. FAQs and Support Common Questions:
  - Debugging Scala? Use REPL, print statements, or IntelliJ debugger.
  - No math background? Focus on intuition; supplemental resources provided.
  - ▶ Other IDEs? VSCode with Metals or text editor with sbt works.

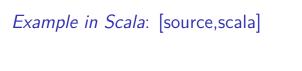
#### Support:

- *TA*: ....
- Discussion Forum: Course portal (link in syllabus)
- Email: Response within 24 hours
- Study Groups: Form via forum or class announcements

[TIP] Engage with peers and ask questions early!

- = Course 1: Mathematical Foundations
- == 10. Course 1 Overview *Topics*:
- ▶ Set theory basics
- Logic fundamentals
- Abstract algebra
- Functions and relations
- Mathematical underpinnings in code

- == 11. Set Theory Basics: What is a Set?
- A set is a collection of distinct objects.
  - Notation: { 1, 2, 3 } a set of numbers
  - Cardinality: The number of elements (e.g.,  $|\{1,2,3\}| = 3$ )
  - ► Order: Sets are unordered (no sequence, unlike lists)
  - Duplicates: No duplicate elements allowed



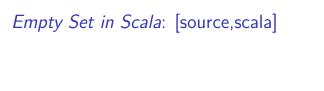
```
val s = Set(1, 2, 3, 2) // Result: Set(1, 2, 3)
```

- == 12. Set Notation & Subsets *Notation*:
- ► A B: A is a subset of B
  - A B: A is a subset of BThe empty set

# Example:

,

If  $A = \{2, 3\}, B = \{1, 2, 3, 4\}$  A B





- == 13. Set Operations
- Union (): Elements in A or B
- ► Intersection ( ): Elements in both A and B
- ▶ Difference (A \ B): In A but not in B
- Complement: Not in set A (relative to a universe U)

Visual: Set Relations + (Two overlapping circles labeled A and B)



```
val a = Set(1, 2, 3) val b = Set(3, 4, 5) a union b // Set(1,2,3,4,5) a intersect b // Set(3) a diff b // Set(1,2)
```

- == 14. Sets in Programming
- Sets are immutable by default in Scala (val s = Set(1,2,3)
- Useful for:
  - Ensuring unique items

    - Simplifying membership checks
    - Mathematical modeling

# Membership Example: [source,scala]

val fruits = Set("apple", "pear", "banana")
fruits.contains("pear") // true fruits.contains("mango") //
false

- == 15. Functions as Mathematical Relations
- A function is a relation mapping each element of set A to
- exactly one element in set B.

  Notation: f : A → B
- $\triangleright$  Example:  $f(x) = x^2$  for x



def sq(x: Double): Double = x \* x // sq(3) = 9.0

- == 16. Injective, Surjective, Bijective
  - Injective (One-to-One): f(a)=f(b) a=b
  - Surjective (Onto): Every element in B is f(a) for some a
  - ▶ Bijective: Both injective and surjective



val nums = Set(1,2,3) val double = nums.map( $\_*2$ ) // Set(2,4,6) - injective

- == 17. Set Theory in Programming Practice
- ► Handling uniqueness: Ensures no duplicate user IDs, transactions, etc.
- ▶ Operations: Filtering, merging datasets, avoiding redundancy
- ► Programming: Set APIs simplify logic over lists or arrays

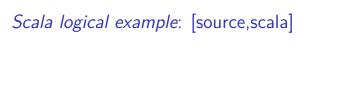
[NOTE] Immutability of sets aligns with functional programming principles.

- == 18. Logic Fundamentals: Propositions & Connectives
- A proposition is a statement that is either true or false
- Connectives:
  - And:
  - Or:
    - Not: ¬
    - Implies:

Example: p: "It rains", q: "I stay home" + If p q, then if it rains, I stay home.

== 19. Truth Tables AND, OR, NOT Example:

p	q	p q	p q	¬р
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true



val rain = true val stayHome = false val both = rain && stayHome // false val either = rain || stayHome // true val notRain = !rain // false

- == 20. Logical Operations in Scala
  - ► Scala uses && for AND, || for OR, ! for NOT
  - Can combine conditions naturally:

[source,scala]

def can Drive(age: Int, has License: Boolean): Boolean = age >= 18 && has License val result = canDrive(20, true) // true

[CAUTION] Logic underpins conditional expressions and error handling.

== 21. De Morgan's Laws

Helps transform logical conditions in coding!



```
// Checking that either x or y is not zero if (!(x == 0 && y
== 0)) \{ ... \} // Equivalent to: if <math>(x != 0 || y != 0) \{ ... \}
```

== 22. Predicate Logic in Programming

► Predicate: Function that returns true/false for inputs

## Example in Scala: [source,scala]

def isEven(x: Int): Boolean = x % 2 == 0

val numbers = List(1,2,3,4) numbers.filter(isEven) // List(2,4)

Predicate logic enables data filtering, validation, searching...

- == 23. Abstract Algebra: Groups, Rings, Fields *Definitions*:
- ► Group: Set + operation (associative, identity element, inverse, closed)
  - ► Ring: Group + a second operation (addition & multiplication, etc.)
- Field: Ring where multiplication has inverses (except 0)

  E.g., Integers form a group under addition; real numbers (excluding

0) form a group under multiplication.

- == 24. Key Algebraic Properties
- - Associativity: (a \* b) \* c = a \* (b \* c)
  - Commutativity: a + b = b + a
  - Identity element: a \* e = a
  - $\blacksquare$  Inverse: a + (-a) = 0

Code Example: [source,scala]

val res1 = (2 + 3) + 4 // 9 val res2 = 2 + (3 + 4) // 9

Associativity in action!

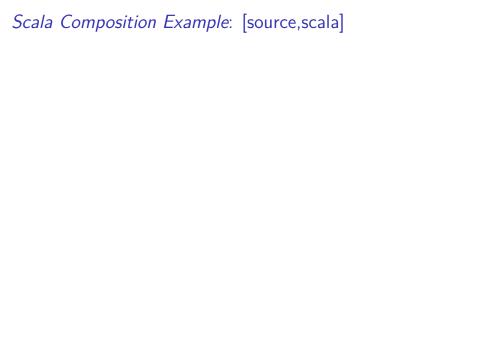
- == 25. Algebraic Principles in Code
- Immutability: Operations return new objects, original remain unchanged
- ▶ Identify operations behaving like algebraic structures:
  - String concatenation: *Monoid* (associative, has identity "")
  - Number addition: *Group*



val s = "Hello" val t = s + "World" val u = t + "" // "" is identity

- == 26. Function Composition
  - ► Compose: Combines two functions (f & g) to create a new function
  - function.

    Notation: (f g)(x) = f(g(x))



```
val f: Int => Int = _ + 1 val g: Int => Int = _ * 2 val fg = f compose g // fg(x) = f(g(x)) fg(3) // 7
```

- == 27. Equivalence Relations
- ► Equivalence relation: relation that is reflexive, symmetric, transitive
- Example: a b mod 3 (a and b leave same remainder when divided by 3)



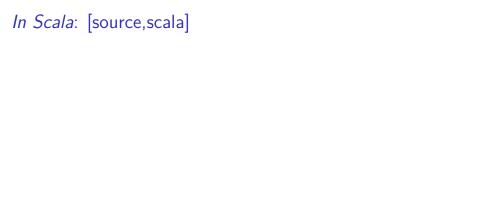
```
def equivMod3(a: Int, b: Int): Boolean = (a \% 3) == (b \% 3) equivMod3(4, 7) // true
```

- == 28. Algebraic Properties in Programming
- ▶ Pure functions: No side effects, always same output for input like mathematical functions
  - Immutability: Operations do not alter original values, matching algebraic structures



val a = List(1,2,3) val b = a.map( $\_$  \* 2) // New list, a unchanged

- == 29. Immutability: A Mathematical Perspective
- ▶ Sets and algebraic structures are immutable by definition
- Functional programming leverages this to ensure predictable, thread-safe code



val original = Set(1,2,3) val added = original + 4 // original remains  $\{1,2,3\}$ , added is  $\{1,2,3,4\}$ 

Immutability is at the heart of functional programming.

- == 30. Functions, Relations, and Immutability
  - ▶ Use relations and functions to model software logic
  - Avoid side effects to preserve functional purity



def square(x: Int): Int = x \* x

[NOTE] Functional purity is closely related to the mathematical definition of functions.

- == 31. Week 1 Recap: Mathematical Foundations
- > Sets and set operations model unique collections
- ▶ Logic is the backbone of code correctness
- Abstract algebraic structures help reason about code organization
- Immutability and pure functions are core to functional design
- ► Mathematical rigor = reliable code

*Next*: Bringing mathematics to life in Scala!

- == 32. Reflect: Where Do You See Mathematics in Code?
  - Think of examples from your own programming where uniqueness, composition, or logical conditions are essential.
- ► Share with the class: How does mathematics clarify programming?

[TIP] Building intuition is key—don't just memorize!

- = Week 2: Introduction to Scala
- == 33. Week 2 Overview Topics:
- Why Scala and functional programming?
- Setting up the environment
- Basic Scala syntax
- ► Values, variables, and types
- Defining and using functions

[NOTE] Scala: modern, concise, powerful

- == 34. Why Scala?
- Hybrid: Combines object-oriented and functional programming
- ► JVM-based: Runs on the Java platform, interoperates with Java libraries
- ► Concise: Fewer lines of code, strong typing
- Popular in: Data engineering, web development, finance, academia
- [NOTE] Major systems: Spark, Akka, Twitter, LinkedIn use Scala

- == 35. Why Functional Programming?
- ▶ Enforces immutability and statelessness
- Reduces bugs, easier to test and reason about
- Promotes modular, reusable code via pure functions
- Handles concurrency more safely

 $[SUCCESS] \ Functional \ code = predictable, \ robust \ software!$ 

== 36. Scala vs. Other Languages

ScalaHybrid $(OO + FP)$ Yes, by defaultYesJavaScript $OO + FP$ No, mutability is defaultNoJavaObject-OrientedNoYesHaskellPurely FunctionalYesNo	Language	Paradigm	Immutability	Runs on JV
	JavaScript Java	OO + FP Object-Oriented	No, mutability is default No	No Yes

Scala blends power, safety, and interoperability.

- == 37. Setting Up Scala Environment
  - 1. Install Java JDK (if needed)

command line/REPL

- 2. Download & install Scala from link https://www.scala-lang.org/download/[scala-lang.org]
- link:https://www.scala-lang.org/download/[scala-lang.org]
- Optional: Install sbt (Scala Build Tool) for project management
   Choose an IDE: IntelliJ IDEA, VSCode (Metals), or use

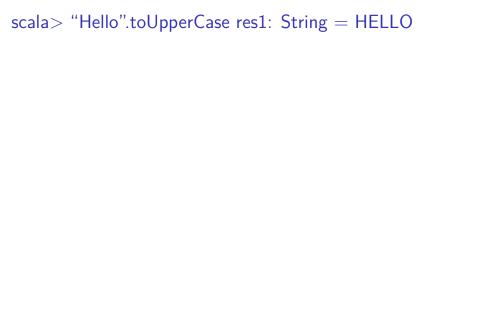
[TIP] Try scala or scalac in a terminal to verify installation.



```
object HelloWorld { def main(args: Array[String]): Unit = { println("Hello, Scala world!") } }
```

- object: Defines a singleton (no instantiation needed)
- ▶ *def*: Defines a function/method

- == 39. Using the Scala REPL
  - ▶ REPL: Interactive Scala shell
  - ▶ Start with scala in your terminal
  - ▶ Type expressions to see results instantly!



- == 40. Basic Scala Syntax
- Statements end with a newline, not necessarily a semicolon
- Curly braces define blocks
- No need for type annotations if type can be inferred

Example: [source,scala]



- == 41. Comments in Scala
  - ▶ Single line: // this is a comment
  - ► Multi-line: [source,scala]

<b>/</b> *	Block	of co	mmen	ts */

Good comments make code easier to understand and maintain!

- == 42. Values (val) vs Variables (var)
  - val: Immutable—cannot be reassigned
  - var: Mutable—can be reassigned



val pi = 3.14 // immutable var count = 0 // mutable count = count + 1 // allowed // pi = 3.15 // error!

[SUCCESS] Prefer val for functional programming!

- == 43. Type Inference
  - ► Scala infers types—no need to specify if obvious
  - Explicit annotation for clarity or required cases



val age = 21 // Inferred as Int val name: String = "Bob" // Explicit String

- == 44. Basic Types in Scala
- ▶ Int: integers
- ▶ Double: floating points
- ▶ Boolean: true/false values
- Char: single characters
- String: sequences of characters



val a: Int = 10 val b: Double = 12.7 val c: Boolean = true val d: String = "Scala"

- == 45. Defining Functions in Scala
  - ▶ def keyword defines a function
  - Explicit input/output types recommended

def add(x: Int, y: Int): Int = x + y

```
val sum = add(2, 3) // 5
```

- == 46. Anonymous Functions (Lambdas)
  - ▶ Inline, unnamed function expressions
  - Use: (input: Type) => expression

val double = (x: Int) => x \* 2 double(3) 
$$//$$
 6

```
val nums = List(1,2,3) nums.map(x => \times * x) // List(1,4,9)
```

- == 47. Higher-Order Functions
  - ► Functions that take other functions as parameters or return them
  - Powerful for map, filter, reduce patterns

Example: [source,scala]

```
def applyTwice(f: Int => Int, x: Int): Int = f(f(x)) applyTwice(x => x + 1, 5) // 7
```

- == 48. Pure vs. Impure Functions
- ► Pure: Output depends only on input; no side effects
- Impure: Relies on/changes external state (I/O, global vars, etc.)

// Pure def square(x: Int): Int = x \* x

```
// Impure var total = 0 def addToTotal(x: Int): Int = \{
total += x; total }
```

- == 49. Blocks and Expression Results
  - ► Scala expressions (including blocks) return values
  - No need for return keyword in most cases



```
def abs(x: Int): Int = \{ \text{ if } (x \ge 0) \text{ x else -x } \}
```



val temp = 30 val weather = if (temp > 25) "Hot" else "Mild" // weather: String = "Hot"

[TIP] Assignment can use result of if-else directly!

- == 51. Pattern Matching Basics
  - Scala's match is like a more powerful switch statement



```
val day = "Mon" val activity = day match { case "Sat" |
"Sun" => "Weekend" case "Mon" => "Back to work"
case _ => "Weekday" } // activity = "Back to work"
```

- == 52. Case Classes
  - Special concise syntax for data containers
  - Immutable, have built-in equals and toString



case class Person(name: String, age: Int) val anna = Person("Anna", 27) val bob = Person("Bob", 32)

- == 53. Immutability in Practice
  - ▶ Default data structures do not change state
  - ▶ Use val and immutable collections



val names = List("Ana", "Ben") val added = names :+
"Carla" // names unchanged, added =
List("Ana", "Ben", "Carla")

[NOTE] Functional code protects against unintended bugs!

- == 54. Scala Collections Overview
  - List: ordered, immutable, allows duplicates
  - > Set: unordered, unique elements
  - ► *Map*: key-value pairs



```
val nums = List(1, 2, 3, 2) // List(1,2,3,2) val s = Set(1, 2, 3, 2) // Set(1,2,3) val m = Map("a" -> 1, "b" -> 2)
```

- == 55. Working with Lists
  - ► Mapping, filtering, reducing, and more



```
val nums = List(1,2,3,4,5) val squares = nums.map(x => x * x) // List(1,4,9,16,25) val evens = nums.filter(\_ % 2 == 0) // List(2,4) val sum = nums.reduce(\_ + \_) // 15
```

- == 56. Map and Set Operations
  - ► *Maps*: Lookup by key
  - > Sets: Membership, union, intersection

## [source,scala]

```
\label{eq:val_capitals} $$ val \ capitals = Map("UK" -> "London", "FR" -> "Paris") $$ capitals.get("UK") // Some("London") \ capitals.get("IT") // None $$
```

```
val s1 = Set(1,2,3); val s2 = Set(2,3,4) s1 & s2 // intersection: Set(2,3)
```

- == 57. Functions on Collections
  - ► Map, filter, fold/reduce for all collections
  - ► Encourages a *declarative* programming style



```
val words = List("cat", "dog", "bird") words.filter(_.length
== 3) // List("cat", "dog")
```

- == 58. Error Handling: Functional Way
  - ▶ Use Option type to represent absence or presence
  - ▶ Use Either for computations that can fail

[source,scala]

def safeDivide(a: Int, b: Int): Option[Int] = if (b == 0) None else Some(a / b)

```
val result = safeDivide(10, 0) // None
```

== 59. Using Either for Errors [source,scala]

```
def parseInt(s: String): Either[String, Int] = try Right(s.toInt) catch
{ case _: NumberFormatException => Left("Not a number") }
```

```
val v1 = parseInt("123") // Right(123) val v2 = parseInt("abc") // Left("Not a number")
```

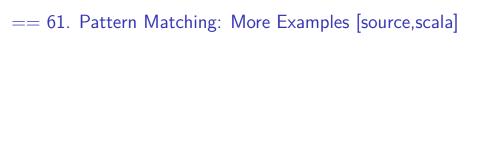
Either lets you return successes or errors without exceptions!

- == 60. Testing Your Scala Code
  - ▶ Use assert for simple checks
  - Use testing frameworks like *ScalaTest*, *Specs2* for bigger projects



def sum(x: Int, y: Int) = x + y assert(sum(2,3) == 5)

[NOTE] Testing is essential for reliable functional code!



def describe(x: Any): String = x match { case 0 =>
 "Zero" case \_: Int => "An integer" case \_: String => "A
 string" case \_ => "Something else" }

Pattern matching works with types, constants, structures...

- == 62. Best Practices in Functional Programming
  - Favor *val*, avoid *var*
  - Prefer pure functions
  - ▶ Use immutable data structures
  - ► Leverage collections' map/filter/reduce
  - ► Handle errors with Option/Either
  - ▶ Aim for predictability and readability

- == 64. Summary of Weeks 1 & 2
  - Explored the *mathematical foundations* of functional programming
  - ▶ Gained hands-on *Scala basics*▶ Learned how immutability, algebra, and logic guide functional
  - Learned now immutability, algebra, and logic guide functional program design
     Ready to build more complex, robust software with these tools!

[SUCCESS] Practice is the key to mastery!

- == 65. Thank You! Next Steps
- Practice basic Scala functions and set operations on your own machine
- ▶ Read more on link:https://docs.scala-lang.org/overviews/scala-book/introduction.html[Scala Book: Introduction]
  - Prepare for next week: Scala collections and more advanced FP patterns

Questions? Reach out anytime!

[NOTE] Happy Functional Coding!