

Scala 2.x

"BRINGING THE BEST OF BOTH WORLDS: OBJECT-ORIENTED AND
FUNCTIONAL PROGRAMMING"

What is Scala?

- Scala is a modern programming language that combines:
 - **Object-Oriented Programming (OOP)**: Classes, objects, and inheritance.
 - **Functional Programming (FP)**: Pure functions, immutability, and expressions.
- Created around 2001 by Martin Odersky.
 - 1st. official release in 2004
 - 1st. stable release in 2006
- The name derives from "Scalable Language", as it works for both small and large projects.
- Runs on the JVM (Java Virtual Machine).
- PS: **Scala is not an extension of Java, but it is completely interoperable with it.**

Why Learn Scala?

Interoperability with Java:

- Seamlessly reuses Java libraries.

Highly expressive: Concise and powerful code.

Popular in Big Data:

- Widely used in tools like Apache Spark and Kafka.

High employability: Increasing presence in startups and major enterprises.

Hybrid paradigms: Learn functional programming without giving up OOP.

Scala vs. Java

Aspect	Scala	Java
Conciseness	Short, expressive code	Verbose
Paradigms	OOP and Functional	Mostly OOP
Immutability	Encouraged	Optional
Syntax	Modern and flexible	Traditional and strict

Key Features of Scala

1. **Strong Static Typing:** Errors caught at compile-time.
2. **Expressions instead of Statements:** More clarity, fewer side effects.
3. **Immutability by Default:** Prevents common bugs.
4. **Interoperability with Java:** Direct integration.
5. **Traits and Rich Class Systems:** Efficient code reuse.

Basic Syntax

```
// Variable declaration  
val immutable = 10 // Immutable value  
var mutable = 20    // Mutable variable  
mutable += 5        // Now mutable = 25
```

```
// Defining a function  
def add(a: Int, b: Int): Int = a + b  
println(add(13, 4)) // Outputs: 17
```

Data Types in Scala

Value Types (Primitive Types):

Scala's primitive types are compatible with Java and represent basic values.

Boolean: true or false.

Byte: 8-bit signed integer (-128 to 127).

Short: 16-bit signed integer (-32,768 to 32,767).

Int: 32-bit signed integer.

Long: 64-bit signed integer.

Float: 32-bit floating-point.

Double: 64-bit floating-point.

Char: A single 16-bit Unicode character.

Unit: Represents no value (similar to void in Java).

Reference Types:

String: Immutable sequence of characters.

Collections: Lists, Maps, Sets, etc.

Scala Collections

- Immutable by default (e.g., List, Set, Map).
- Mutable collections available when necessary (e.g., ArrayBuffer, MutableList).
- Rich API for operations like map, filter, reduce.
- Support for both sequential and parallel processing.

```
// Lists: Ordered and immutable.
val fruits = List("Apple", "Banana", "Cherry")

// Arrays: Fixed-size and mutable.
val numbers = Array(1, 2, 3, 4)

// Sets: Unordered and unique elements.
val uniqueNumbers = Set(1, 2, 3, 3)
println(uniqueNumbers) // Outputs: Set(1, 2, 3)

// Maps: Key-value pairs.
val ages = Map("Alice" -> 25, "Bob" -> 30)
```


Operators in Scala

Arithmetic Operators

Addition: +

```
val sum = 5 + 3 // 8
```

Subtraction: -, Multiplication: *, Division: /, Modulus: %

Relational Operators

Equals: ==, Not Equals: !=

```
val isEqual = 5 == 5 // true
```

Greater Than: >, Less Than: <, etc.

Logical Operators

AND: &&, OR: ||, NOT: !

```
val isTrue = (5 > 3) && (4 < 6) // true
```

Assignment Operators

Assign: =, Add and Assign: +=, Subtract and Assign: -=

```
var x = 10  
x += 5 // x = 15
```

Special Operators

Method Invocation: Operators are methods:

```
val sum = 5.+(3) // Equivalent to 5 + 3
```

Precedence: Operators follow the method precedence rules.

Object-Oriented Programming in Scala

Defining a class:

```
class Person(val name: String, val age: Int) {  
  def greet(): String = s"Hello, my name is $name and I am $age years old."  
}  
val person = new Person("John", 30)  
println(person.greet()) // Outputs: Hello, my name is John and I am 30 years old.
```

Using traits (similar to interfaces with implementation):

```
trait Greetable {  
  def greet(): Unit = println("Hello!")  
}  
class Friend extends Greetable  
val friend = new Friend  
friend.greet() // Outputs: Hello!
```

Control Structures in Scala

Scala uses expressions for conditional branching.

Syntax Example:

```
val number = 10
if (number > 0) {
  println("Positive number")
} else if (number < 0) {
  println("Negative number")
} else {
  println("Zero")
}
```

Expression Example:

```
val result = if (number > 0) "Positive" else "Non-positive"
println(result) // Outputs: Positive
```

Loops in Scala

while Loop: Repeats while a condition is true.

```
var count = 0
while (count < 5) {
  println(count)
  count += 1
}
```

do-while Loop: Executes at least once, then checks the condition.

```
var count = 0
do {
  println(count)
  count += 1
} while (count < 5)
```

for Loop: Iterates over collections or ranges.

```
for (i <- 1 to 5) { println(i) }
// Exclusive range:
for (i <- 1 until 5) { println(i) }
```

for with Filters

```
for (i <- 1 to 10 if i % 2 == 0) {
  println(i) // Outputs: 2, 4, 6, 8, 10
}
```

for Yield: Produces a new collection.

```
val squares = for (i <- 1 to 5) yield i * i
println(squares) // Outputs: Vector(1, 4, 9, 16, 25)
```

Functional Programming in Scala

Higher-order functions:

```
def applyTwice(f: Int => Int, x: Int): Int = f(f(x))  
val result = applyTwice(x => x + 2, 5)  
println(result) // Outputs: 9
```

Immutable collections:

```
val nums = List(1, 2, 3, 4)  
val sum = nums.reduce(_ + _)  
println(sum) // Outputs: 10
```

Pattern matching:

```
val number = 2  
number match {  
  case 1 => println("One")  
  case 2 => println("Two")  
  case _ => println("Other")  
}  
// Outputs: Two
```

Advanced Code Examples

Defining and using case classes:

```
case class Point(x: Int, y: Int)
val point = Point(3, 4)
println(point) // Outputs: Point(3,4)
```

Lazy evaluation:

```
lazy val expensiveComputation = {
  println("Computing...")
  42
}
println("Before accessing lazy value")
println(expensiveComputation) // Computes and prints: 42
println(expensiveComputation) // Directly prints: 42
```

Scala tools

Scala Compiler: scalac

Scala REPL: Test and experiment with code interactively.

Build Tools: SBT (Scala Build Tool), Maven, Gradle.

Scala playground: Scaltie (<https://scastie.scala-lang.org/>)

Scala frameworks

- Big Data: Core language for Apache Spark.
- Web Development: Frameworks like Play! Framework.
- Microservices: Lagom, Lift.
- In-demand tools: Kafka, Akka, etc.
- Query and access databases in Scala syntax: Slick

Challenges with Scala

Learning curve: Mixed paradigms can be challenging for beginners.

Smaller community: Fewer beginner resources compared to Java.

Code complexity: Overuse of features can lead to unreadable code.

Conclusion

Scala is a versatile and powerful language that blends OOP and FP.

Ideal for students interested in Big Data, scalable systems, and modern programming paradigms.

Learning Scala opens doors to diverse job opportunities in high-demand fields.

...

Questions?

Getting Started

1. Install Scala: <https://www.scala-lang.org/>
2. Set up an IDE: IntelliJ IDEA or VS Code with Scala plugins.
3. Explore the REPL to experiment with code.
4. Practice on Scala Exercises: <https://www.scala-exercises.org/>.
5. Read the official documentation and start small projects.

OR (for some simple tests)

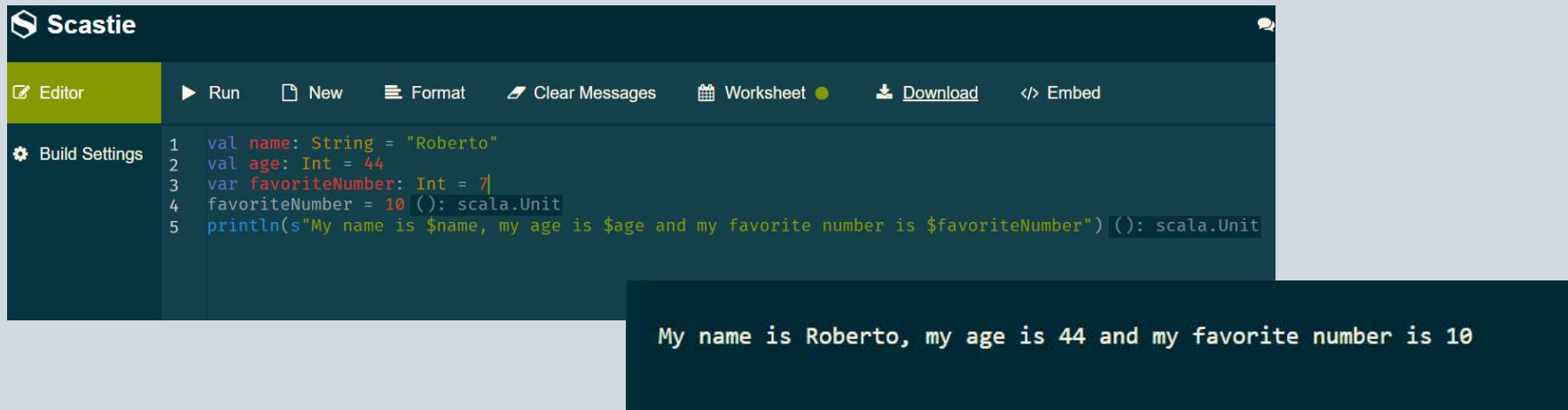
1. Scala playground: Scaltie (<https://scastie.scala-lang.org/>)

Exercise: Variables and Data Types

Define variables:

- One immutable (val) as a String Type with your name
- One immutable (val) as a Integer Type with your age.
- One mutable (var) with your favorite number.

Modify the mutable variable and print the values.



The screenshot shows the Scastie web interface for running Scala code. The left sidebar has a 'Build Settings' section. The main editor area contains the following Scala code:

```
1 val name: String = "Roberto"
2 val age: Int = 44
3 var favoriteNumber: Int = 7
4 favoriteNumber = 10 (): scala.Unit
5 println(s"My name is $name, my age is $age and my favorite number is $favoriteNumber") (): scala.Unit
```

Below the code editor, a dark box displays the output of the program:

```
My name is Roberto, my age is 44 and my favorite number is 10
```

Exercise: Conditional Statements

- Write a program that checks whether a number is positive, negative or zero.
- Use an if/else block to implement the logic.
- Create this logic inside a function that
 - Receive the number as a parameter
 - Print, in the end, the number and if it is positive, negative or zero

```
val x = 2
val result =
  if (x == 1) "Y"
  else if (x == 0) "N"
  else "N/A"
print(result) (): scala.Unit
```

```
val x = 2

def add5(value: Int): Unit = {
  | print(value + 5)
  | }

add5(x) (): scala.Unit
```

Exercise: Loops

Use a for loop to print all even numbers from 1 to 20.

After:

- Modify the program to also calculate the sum and the average of these numbers.

```
for (i <- 1 to 10 if i % 2 == 0) {  
  println(i) // Outputs: 2, 4, 6, 8, 10  
}
```

```
var sum = 0  
for (i <- 1 to 20 if i % 2 == 0) {  
  println(i)  
  sum += i  
}  
println(s"Sum: $sum")
```

Exercise: Collections and Functions

Create a list of integers from 1 to 10.

Use the following functions:

- map: Multiply each number by 3.
- filter: Keep only odd numbers.
- reduce: Find the sum of the list.

In Scala, the underscore `_` is a powerful and versatile symbol used in several contexts, each with its specific meaning.

For this case, it is used as a placeholder for a parameter:

```
val numbers = List(1, 2, 3, 4, 5)
val doubled = numbers.map(_ * 2) // Equivalent to: numbers.map(x => x * 2)
println(doubled) // Output: List(2, 4, 6, 8, 10)
```

```
val numbers = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
val doubled = numbers.map(_ * 2)
val evens = numbers.filter(_ % 2 == 0)
val sum = numbers.reduce(_ + _)
println(doubled)
println(evens)
println(sum)
```

Contact

Roberto Passos Rodrigues Jr.

Developer / Data Engineer / Tech Lead at Natixis

Email: robertoprjr@gmail.com

GitHub: <https://github.com/robertoprjr>

LinkedIn: <https://www.linkedin.com/in/roberto-passos-rodrigues-jr/>