Scala

Jugal Gajjar
Saptorshee Nag
Kaustik Ranaware
Saurabh S R
Michael Womack

Introduction

Scala is a high-level, multi-paradigm programming language that combines object oriented programming and functional programming.

Popular use cases for scala are big data, distributed computing, micro services, ad-serving, communication platforms, real-time information processing, and so ons

Scala was designed to be high scalable making it great for running small scripts or large distributed systems

It runs on the Java Virtual Machine (JVM), making it compatible with existing Java Libraries

Contributors:

- Scala Center, Lightbend, and VritusLab

Names, Binding, and Scopes

Scala allows you to use operator-like symbols as identifiers, which is not common in many other languages.
Example: +, :, ::, and =>

Scala allows defining methods and variables using operator-like symbols, which makes it easier to define custom operators. For instance, you can define a method as

def + (that: MyClass): MyClass = {...}

A unique feature of Scala is the **companion object**. When you define a class and an object with the same name, the object is a "companion" to the class. They share the same name and scope but can access each other's private members.

Names, Binding, and Scopes contd..

Example of companion class:

```
class MyClass {
  private val secret = "Hidden"
}
object MyClass {
  def reveal = (new MyClass).secret // Can access 'secret' because it shares the same scope
}
```

Data Types

Scala’s data types are versatile, combining the rigor of static typing with the expressiveness of functional programming.

In Scala, **everything is an object** — even primitive types like Int, Double, and Boolean.

Scala has special bottom types, Nothing and Null.
def fail(message: String): Nothing = throw new Exception(message)

Scala provides a powerful way to handle the absence of values via the Option type. This is a unique and functional alternative to null.

Data Types contd...

Scala has **tuples**, They are immutable and can store multiple items of potentially different types without needing to create a specific class or case class. A tuple can hold from 2 to 22 elements, each can be different data type.

```
val tuple = (1, "Scala", 3.14)                                  // Tuple of Int, String, and Double
```

Scala introduces **case classes**, which are a special type of class optimized for immutability and pattern matching.

```
case class Person(name: String, age: Int)
val person = Person("John", 30)
person.name                                                     // Accessing fields
```

Expressions and Assignment Statements

Expressions

Everything is an Expression: In Scala, almost everything (e.g., if statements, match cases, loops) is treated as an expression, meaning they return a value.

Example: val x = if (a > b) a else b

The above is an expression where if-else returns the larger of a or b.

Functional Nature: Scala encourages a functional programming style. Since expressions return values, you can often avoid using mutable state.

Type Inference: Scala's compiler automatically infers the type of expressions, reducing the need for explicit type annotations.

Example: val y = 10 // The type of y is referred to be Int.

Expressions and Assignment Statements contd….

Assignment Statements:

Immutable by Default: Variables in Scala are immutable by default, declared using val. Once assigned, their value cannot change.

Example: val z = 42
z = 50 // **This will result in a compilation error.**

Mutable Variables: If you need to reassign a variable, use var. However, this is generally discouraged in favor of immutability.

Example: var counter = 0
counter = counter + 1 // **Reassigning a new value to the mutable variable.**

Expressions vs. Statements: Scala emphasizes expressions over statements (which do not return values). Even loops (e.g., for, while) can be considered expressions, although they are usually avoided in functional programming in favor of more abstract constructs like map, flatMap, etc.

Scala and Object Oriented Programming

Due to Scala running on top of the Java Virtual Machine, Scala’s object orient programming supports the following features:

- Classes
- Objects
- Inheritance
- Traits
- Encapsulation
- Polymorphism

Unlike Java, classes aren’t required allowing scala programs to integrate functional programming languages into their applications seamlessly without disrupting the project structure.

Concurrency

Concurrency is a key aspect of the language, achievable via it’s several tools and abstractions:

- Java Threads: Scala runs on the Java virtual machine meaning it inherits all of Java’s threading libraries (i.e Execution Context)
- Futures and Promises: A part of Scala’s standard library; Future represents a value that may not be available yet, allowing for non-blocking operations. “Promise” can be used to complete a future what the value is known
- Parallel Collections: A series of included operations; map, filter, and reduce. Automatically parallelizing code written via threads.
- Akka Streams: Scala’s native reactive programming libraries that allow developers to work with multiple streams of asynchronous data at a time

Scala has high levels of abstraction that simplify concurrency for the developer, yet it is not immune to race conditions.Thus using immutable data types is encouraged.

Event Handling

Event handling in Scala is commonly achieved through its actor-based model and reactive programming frameworks like Akka. By utilizing actors for asynchronous message passing, Scala efficiently handles events in a concurrent, non-blocking way.

Reactive Programming: Scala offers good support for reactive programming, making it suitable for handling asynchronous events.

Akka Framework: One of the most common frameworks for event-driven programming in Scala is Akka. It provides an actor-based model to handle concurrency and event handling in a scalable and resilient way.

Scala Swing (for GUI event handling): If you are working with GUIs, Scala provides the Scala Swing library, where you can handle UI events like button clicks, window resizing, etc.

Event Handling Contd....

Actor Model: Actors are objects that handle communication asynchronously by exchanging messages. Each actor has its own state and behavior and interacts with other actors via message passing.

Example:

```
import akka.actor._
class MyActor extends Actor {
  def receive = {
    case "Hello" => println("Hello back!")
    case _ => println("Unknown message")
  }
}
val system = ActorSystem("MySystem")
val myActor = system.actorOf(Props[MyActor], name = "myActor")
myActor ! "Hello"
```

Here, the actor **MyActor** reacts to the message "Hello" by printing a response.

Exception Handling

- **Basic Exception Handling**: Use **try, catch, and finally**

  try: The code that may throw an exception is placed inside the try block.
  catch: You can have multiple case statements to handle different types of exceptions.
  finally: This block will always execute, regardless of whether an exception was thrown or caught.

```
try { // Code that might throw an exception
} catch { case e: ExceptionType1 => // Handle ExceptionType1
          case e: ExceptionType2 => // Handle ExceptionType2
} finally { // Code that always runs (optional) }
```

Exception Handling

- **Using Try**: The ***Try*** type is a more functional way to handle exceptions. It can be either a Success or a Failure, making it easier to work with error handling without using exceptions directly.

  ```
  import scala.util.{Try, Success, Failure}
  val result: Try[ReturnType] = Try { // Code that might throw an exception }
  result match {
                  case Success(value) => // Handle the success
                  case Failure(exception) => // Handle the failure case
  ```

  ```
  Eg.  val result: Try[Int] = Try(10 / 2) // Success(5)
       val result2: Try[Int] = Try(10 / 0) // Failure(ArithmeticException)
                  }
  ```

- **Throwing Exceptions(throw)**: You can throw exceptions explicitly using the ***throw*** keyword, allowing for custom error handling.

Functional programming in Scala

Functional Programming focus on ***What to do*** instead of ***How to do it*****,** similar to the declarative style.

Scala along with Object-Oriented also supports **Functional Programming.**

Functional programming in Scala focuses on :

1. **Pure functions**
2. **Immutability**
3. **Higher-order functions**
4. **First-Class Functions**
5. **Function Composition**

Pure Function vs Impure Function

Pure Function :

```
def add(a: Int, b: Int): Int = a + b
println(add(2, 3)) // Output: 5
println(add(2, 3)) // Output: 5
```

Why pure?

No side effects.

Only depends on a and b.

Gives the same O/P for the same I/P.

Impure Function :

```
var result = 0 // External mutable state
def impureAdd(a: Int, b: Int): Int = {
        result = a + b //
        result }
println(impureAdd(2, 3)) // Output: 5
println(impureAdd(2, 6)) // Output: 8
println(result) // Output: 8 (State has changed)
```

Why Impure?

The function modifies the **result** variable every time it is called, leaving a **side effect**.

Immutability

Immutability in Scala

```
// Immutable variable using val
val numbers = List(1, 2, 3, 4, 5)

// Trying to modify it will result in a compilation error
numbers = List(6, 7, 8) // Error: reassignment to val.
```

Scala provides several built-in immutable collections, such as ***List, Set, Map*** and ***Vector***.

Higher Order Function

Higher Order Function in Scala:

```
// Higher-order function that takes a function as an argument
def applyFunction(f: Int => Int, value: Int): Int = f(value)

// Lambda function to square a number
val square: Int => Int = x => x * x

println(applyFunction(square, 5)) // Output: 25
```

A **higher-order function** is a function that takes **another function** as an argument or returns a function as a result.
In this case, **applyFunction** is a **higher-order function** because it accepts another function f of type Int => Int.

First-Class Functions

Functions are treated as **First-Class Citizens**

1. Passed as argument

```
def Funct(f: Int => Int, value: Int): Int = f(value)
val square: Int => Int = x => x * x
val squaredVal = Funct(square, 4)
```

2. Assigned to variables

```
val add: (Int, Int) => Int = (a, b) => a + b
 val result = add(3, 5) // result is 8
```

3. Returned from other function

```
def adder(addedValue: Int): Int => Int = { (x: Int) => x + addedValue }
val addThree: Int => Int = adder(3)
val resultAddThree = addThree(5)
```

Questions?

Our Project - Big Data Processing Pipeline

Project Objective

- To build an end-to-end big data processing pipeline using Scala, capable of efficiently ingesting, storing, processing, and analyzing large datasets to generate actionable insights.
- Enable real-time data processing for fast decision-making, and ensure scalability and fault-tolerance for handling large-scale data.

Constraints

1. Data Volume
2. Data Consistency
3. Latency
4. Scalability
5. Resource Management

Features

1. Data Ingestion
2. Data Storage
3. Data Processing
4. Data Transformation
5. Data Analysis
6. Logging and Monitoring

Technology That Will Be Used

1. Scala
2. Apache Kafka
3. Apache Spark
4. HDFS (Hadoop Distributed File System)
5. Spark SQL/DataFrames

Use Cases

1. E-Commerce
2. Healthcare
3. Finance
4. Telecommunications
5. Internet of Things (IoT)

Thank You!