Anshu Reddy Ashanna
Naiyani Paladugu
Krithika Reddy Sangireddy
Kush Palkesh Dudhia

INTRODUCTION

- Scala is a versatile, high-level programming language that combines object-oriented and functional programming paradigms.
- It runs on the Java Virtual Machine (JVM) and is designed to be concise, expressive, and scalable.
- Scala addresses some limitations of Java while maintaining standards with Java libraries.
- Scala is widely used for building high-performance systems in big data processing, web services, and machine learning, with companies like Twitter, Netflix, and LinkedIn leveraging it for their backend infrastructure.

NAMES, BINDING, SCOPES

- In Scala, like many other programming languages, names refer to **identifiers** which are used to specify various program elements such as **variables, classes, objects, methods, and functions.**
- **Operator Identifiers**: Scala allows the use of symbols as valid identifiers, which is not common in many languages. For example, +, ++,=, and :: are valid names for methods or functions in Scala.
- Example :

```
class Vector(val x: Int, val y: Int) {
        def +(other: Vector) = new Vector(x + other.x, y + other.y)
  }
val v1 = new Vector(1, 2)
val v2 = new Vector(3, 4)
val result = v1 + v2  // Uses the + operator defined in the Vector class
```

NAMES, BINDING, SCOPES

- Scala allows you to use reserved keywords as variable names by enclosing them in **backticks**.
- Example : val `class` = "this is valid"

  println(`class`)

- In Scala, **singleton objects** are used to define a single instance of a class. Unlike Java's static members, Scala does not allow static methods or variables inside a class. Instead, Scala uses singleton objects to hold static-like members and methods.
- Example : object MathHelper {

  def square(x: Double): Double = x * x

  }

NAMES, BINDING, SCOPES

- A **companion object** is a singleton object that has the same name as a class and is defined in the same file. A class and its companion object can access each other's private members.
- Example: class Person(val name: String, val age: Int)

  object Person {

  def apply(name: String): Person = new Person(name, 0) }

- **Lazy binding:** Scala supports lazy evaluation of values using the lazy keyword. A lazy val is not evaluated until it is first accessed, providing a form of deferred or lazy initialization.
- Example: lazy val expensiveComputation = {

  println("Computing...") 42 }

NAMES, BINDING, SCOPES

- In Scala, **scope** defines the visibility and lifetime of variables, methods, and objects. Scala has similar scoping rules as other languages (local, class-level, and global scope).
- **Nested Functions:** Scala allows to define functions within functions, giving inner functions access to variables from the outer function's scope.
- Example: def outer(x: Int): Int = {

```
    def inner(y: Int): Int = x + y

   inner(10)

}
```

DATA TYPES

- In Scala, the data types are similar to Java in terms of length and storage.
- All data types are objects, even what are considered primitive types in other languages (like Int, Boolean, and Double).
- **Unit** is similar to void in other languages. It's used when a function doesn't return a meaningful value.
- Example: def printHello(): Unit = println("Hello")
- **Option** type is used to represent optional values. It can either be Some(value) or None to handle absence of values safely.
- Example : val someVal: Option[Int] = Some(5)

  val noneVal: Option[Int] = None

DATA TYPES

- **Nothing** is a subtype of all types and represents an absence of value, often used for methods that never return (e.g., throw).
- Example: def infiniteLoop(): Nothing = throw new Exception("Never returns")
- **Any** is the supertype of all types, while **AnyRef** is the supertype of all reference types.
- Example : val anyValue: Any = 42

```
val anyRefValue: AnyRef = "Hello"
```

- **Case classes** in Scala are special types of classes that come with built-in features like immutability, pattern matching, and auto-generated methods (toString, equals, hashCode).
- Example: case class Person(name: String, age: Int)

```
val person = Person("Alice", 30)

println(person.name)  // Outputs: Alice
```

EXPRESSIONS

- Expressions are the main components of a Scala program that evaluate to a value.
- Expressions exist at compile-time for type checking, but they evaluate to values at run-time.
- **Example:**

```
val x = 2          // x is an expression that evaluates to the value 2
```

- A value is information stored in the computer's memory. It exists at run-time.
- **Types Of Expression:**

  a)Literal Expression

  b)Compound Expression

LITERAL EXPRESSION

- A literal expression represents a fixed value that stands “for itself”.
- **SYNTAX:**

  42 // A literal expression

- **EXAMPLE:**

  val num = 42 // res0: Int = 42

- The literal 42 is the textual representation, while the value is what exists in memory after evaluation

COMPOUND EXPRESSIONS

a) **CONDITIONAL**

- It allows us to choose which expression to evaluate based on a condition
- **SYNTAX:**

  if (condition) trueExpression else falseExpression

- **EXAMPLE:**

  val result = if (1 < 2) "Yes" else "No"

  // result: String = "Yes"

COMPOUND EXPRESSIONS

b) BLOCKS

- Blocks are expressions that allow us to sequence computations together.
- **SYNTAX:**

```
{
    declarationOrExpression …
    Expression
}
```

- **EXAMPLE:**

```
val blockResult = {
    val a = 10
    val b = 20
    a  + b  // This is the final expression in the block
 }  // blockResult: Int = 30
```

Support to Object Oriented Programming

- Scala is a hybrid between functional and object-oriented programming, and it smoothly integrates the features of object-oriented and functional languages.
- Object-oriented programming concepts in Scala:
  a) Classes and Objects
  b) Encapsulation
  c) Inheritance
  d) Polymorphism
  e) Traits

CLASSES AND OBJECTS

1. **Defining Classes:**

   Example: class Bread

2. **Creating Objects:**

   Example: val whiteBread= new Bread

3. **Defining Fields:**

```
class Bread {
        val name: String = "white“
        var weight: Int = 1
}
```

ENCAPSULATION

- It describes the idea of bundling data and methods that work on that data within a class.
- **Example:**

```
class Sandwich(bread: Bread, filling: ArrayBuffer[String]) {
        private def getFillingsName: String = {
                filling.mkString(", ")
        }
}
```

INHERITANCE

- **Inheritance** is a mechanism where one class can inherit the properties and behaviors of another class.
- This allows for code reuse and the creation of more specialized classes based on more general ones.
- Drawback:

  Does not support Multiple Inheritance

Example:

```
class Animal {
   def speak(): Unit = {
      println("Animal makes a sound")
    }
}
class Dog extends Animal {
      override def speak(): Unit = {
            println("Dog barks")
      }
}
```

POLYMORPHISM

- The ability of different types (classes) to be treated as if they were the same type, typically through inheritance or interfaces (traits in Scala).
- There are two main types of polymorphism in Scala:

a) **Method Overloading:** Overloading a method means using a method name but applying different logic, depending on the parameters defined in the methods.

b) **Method Overriding:** Overriding is used in a subclass to redefine the implementation of a method provided by its superclass.

TRAITS

- A **trait** is used to define an object is created as a mixture of methods that can be used by different classes without requiring multiple inheritances.
- Traits can have both **abstract** and **non-abstract** methods, fields as its members. When you do not initialize a method in a trait, then they are abstract, while the ones that are initialized are called non-abstract.
- **SYNTAX:**

```
trait Trait_Name{

    // Variables
   // Method

}
```

CONCURRENCY

When developing parallel and concurrent applications in Scala, you can utilize the native Java Thread. However, the Scala Future provides a more high-level and idiomatic solution, making it the preferred choice

“A Future represents a value which may or may not currently be available, but will be available at some point, or an exception if that value could not be made available.”

For example, we call a method wrapped in a Future, the main thread proceeds to execute the following code while the Future processes the task in the background. This **non-blocking behavior** is essential for creating responsive applications, particularly in situations like web services or user interfaces, where delays can negatively impact the user experience.

SOME FEATURES OF CONCURRENCY

Promising Non-Blocking: Using Futures enables non-blocking behavior, allowing the main thread to continue executing while waiting for tasks to complete.

Composability: **Futures** can be easily composed using methods like **map, flatMap** making it simple to chain asynchronous operations.

Error Handling: Futures provide mechanisms for handling exceptions, enabling developers to specify fallback behavior using methods like **recover.**

Parallel Collections: Scala offers parallel collections that allow developers to easily parallelize operations on collections, making it easier to take advantage of multi-core processors.

Simplicity and Readability: The concurrency model in Scala is designed to be more intuitive and easier to read than traditional thread management in Java.

Thread Safety: Scala encourages immutability and functional programming principles, which help in writing thread-safe code by minimizing shared mutable state.

Integration with Java: Scala concurrency features integrate well with Java concurrency utilities, allowing developers to leverage existing Java frameworks and libraries.

INPUT

```scala
import scala.concurrent.{Future}
import scala.concurrent.ExecutionContext.Implicits.global
import scala.util.{Success, Failure}

object FactorialFutureExample extends App {

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

  // Wrap the factorial calculation in a Future
  val number = 5
  val futureFactorial = Future {
    factorial(number)
  }

  println(s"Calculating factorial of $number...")

  // Handling the result of the Future
  futureFactorial.onComplete {
    case Success(result) => println(s"The factorial of $number is $result")
    case Failure(e) => println(s"An error occurred: ${e.getMessage}")
  }

  // Keep the main thread alive to allow time for the Future to complete
  Thread.sleep(5000)  // Adjust the sleep time as needed
}
```

OUTPUT

```
Calculating factorial of 5...
The factorial of 5 is 120
```

CODE EXPLAINED

- This program showcases the use of **Future** for asynchronous computation by calculating the factorial of a specified number, in this case, 5. It defines a recursive **factorial** function that returns the factorial value. By wrapping this computation in a **Future**, the program allows the calculation to run in the background without blocking the main thread, enabling the application to remain responsive. While the factorial is being computed, a message is displayed to indicate that the calculation is in progress.
- The program employs the **onComplete** , method to manage the result of the **Future**, either printing the calculated factorial or an error message if an exception occurs. To ensure the main thread stays alive long enough for the computation to finish, a **Thread.sleep(5000)** call is included.
- This example effectively illustrates the advantages of using **Future** in Scala for concurrent tasks, allowing for efficient and responsive applications

Exception and Event Handling

- **All exceptions are unchecked** (no concept of checked exceptions).
- Use try { ... } catch { ... } block similar to Java.
- **Catch block** uses **pattern matching** to identify and handle exceptions.

Catching Exceptions

- Scala allows you to **try/catch** any exception in a single block and then perform pattern matching against it using **case** blocks.

Throwing Exceptions

- Throwing an exception looks the same as in Java. You create an exception object and then you throw it with the **throw** keyword as follows.
  Example: throw new IllegalArgumentException

Functional Programming

Higher-order Functions:

- **Definition**: Functions that either **accept functions** as arguments or **return functions** as results.
- **Key Features**:
  - Can accept **one or more parameters** that are functions.
  - Offer **flexibility** in programming.
- **Use Case**: Higher-order functions allow modular, reusable code by leveraging function arguments or results.

Currying in Scala:

- **Definition**: The process of transforming a function with multiple arguments into a series of **one-argument functions**.
- **Application**: Commonly used in various **functional programming languages**.
- **Benefit**: Makes functions more flexible and easier to reuse in different contexts.

Functional Programming

Pure Functions:

- **Definition**: Always return the same result for the same inputs.
- **Key Characteristics**:
  - No **side effects** (do not modify external state).
  - Depend **only on input parameters**.
- **Example**: A function that computes and returns a result based solely on the input.

Non-pure Functions:

- **Definition**: May change internal or external state (allow context changes).
- **Characteristics**:
  - Can have **side effects** (e.g., modifying global variables, updating files, or databases).
  - Results may vary depending on external factors.
- **Use Case**: Necessary when actions need to affect state (e.g., file writing, database updates).

Functional Programming

- Defines a function factorial(n) to compute the factorial of an integer n.
- If n == 0, the function returns **1**.
- This is because **0! = 1**
- If n != 0, the function calls itself recursively with (n-1).
- Multiplies the result by n (i.e., n * factorial(n-1)).

```
def factorial(n: Int): Int = {
  if (n == 0) 1
  else n * factorial(n-1)
}
```

- The recursion continues until n reaches **0**.
- At that point, the recursion stops, and the final factorial result is returned.

Note that in Scala, the return keyword is optional if the last expression in the function body is the return value.

PROJECT DESCRIPTION
TASKMATE

- **OBJECTIVE:**

  To create a web application for tracking daily tasks and activities, allowing users to view their lists and stay organized.

  The Scala **TASKMATE** web application is a project that aims to create a simple and intuitive task management tool for users.

  The application is built using the Scala programming language and utilizes the Play Framework to provide a robust and scalable web application architecture.

PROJECT DESCRIPTION

- **LIMITATIONS:**
  - Calendar integration is not allowed.
  - Time logging is prohibited.
  - No external File Uploads

- **TECHNOLOGIES USED:**
  - Scala3
  - SQL
  - Play Framework

PROJECT DESCRIPTION

- **FEATURES:**
- Login & Register User
- Add, modify, or delete a task.
- Each task will include a checklist for marking completion.
- Adding Comments section to help remind any special thing for a particular event.