Scala Day 2

Scala Programming.

By Dhandapani Yedappalli Krishnamurthi Oct 7, 2025

🧩 1. Immutable vs Mutable Collections

🔹 Explanation

- **Immutable collections** cannot be changed after creation — any modification creates a new collection.
- **Mutable collections** can be updated in place.

Type	Package	Examples
Immutable (default)	scala.collection.immutable	List, Vector, Map, Set
Mutable	scala.collection.mutable	ArrayBuffer, ListBuffer, HashMap, Set

✅ Example

```
object CollectionExample extends App {
  // Immutable List
  val fruits = List("Apple", "Banana", "Orange")
  val newFruits = fruits :+ "Grapes"  // creates a new list
  println(s"Original: $fruits")
  println(s"New: $newFruits")
  // Mutable ListBuffer
  import scala.collection.mutable.ListBuffer
  val numbers = ListBuffer(1, 2, 3)
  numbers += 4  // modifies in place
  println(s"Mutable ListBuffer: $numbers")
}
```

⚙️ ETL Context

Immutable structures are safer for parallel transformations (e.g., Spark RDDs). Mutable ones are useful for local buffering before writing to files.

🧠 2. Higher-Order Functions (HOFs)

🔹 Explanation

Functions that take other functions as arguments or return them as results.

✅ Example

```
object HOFExample extends App {
  def processList(nums: List[Int], func: Int => Int): List[Int] = {
```

```
  nums.map(func)
 }
 val nums = List(1, 2, 3, 4, 5)
 val doubled = processList(nums, x => x * 2)
 println(doubled)
}
```

⚙️ ETL Context

Used in transformations like .map(), .filter(), .reduce() on Spark RDDs and DataFrames.

🧮 3. Currying and Partial Functions

🔹 Currying

Breaking down a function of multiple arguments into a series of single-argument functions.

```
object CurryingExample extends App {
 def multiply(x: Int)(y: Int): Int = x * y
 val timesTwo = multiply(2)_
 println(timesTwo(10))  // 20
}
```

🔹 Partial Function

A function that is **not defined for all inputs**.

```
object PartialExample extends App {
 val divide: PartialFunction[Int, Int] = {
  case x if x != 0 => 100 / x
 }
 if (divide.isDefinedAt(0)) println(divide(0))
 else println("Division by zero not allowed")
}
```

⚙️ ETL Context

Currying helps build reusable parameterized functions (e.g., loggers, connectors).
Partial functions help handle invalid data conditions gracefully.

🔁 4. For-Comprehensions

🔹 Explanation

A readable syntax for chaining map, flatMap, and filter.

```
object ForComprehensionExample extends App {
 val data = List(10, 20, 30)
 val result = for {
  x <- data
  if x > 15
 } yield x / 10
 println(result) // List(2, 3)
}
```

⚙️ ETL Context

Used for transforming hierarchical or filtered data flows.

⚡ 5. Anonymous and Lambda Functions

🔹 Explanation

Anonymous (lambda) functions are unnamed functions often used inline.

```
object LambdaExample extends App {
 val numbers = List(1, 2, 3, 4)
 val squares = numbers.map(x => x * x)
 println(squares)
```

```
}
```
or even shorter:
```
val squares = numbers.map(_ * _)
```
⚙️ ETL Context

Widely used in Spark transformations (df.filter(_ > 10)).

💡 6. Implicits and Type Parameters

🔹 Implicits

Allow automatic conversions or values to be passed without explicitly mentioning them.
```
object ImplicitExample extends App {
  implicit val defaultTax: Double = 0.1
  def calculateTotal(amount: Double)(implicit tax: Double): Double = amount + (amount * tax)
  println(calculateTotal(100))  // uses implicit 0.1
}
```
🔹 Type Parameters (Generics)

Used for defining generic methods or classes.
```
object GenericExample extends App {
  def printList[T](list: List[T]): Unit = list.foreach(println)
  printList(List("Scala", "Python", "Java"))
}
```
⚙️ ETL Context

Implicits simplify configuration injection; generics make reusable data processors.

🧱 7. Object-Oriented Scala (Classes, Traits, Inheritance)
```
trait Logger {
  def log(msg: String): Unit = println(s"[LOG]: $msg")
}
class ETLJob(name: String) extends Logger {
  def run(): Unit = {
    log(s"Running ETL job: $name")
  }
}
object OOPExample extends App {
  val job = new ETLJob("DailySalesLoad")
  job.run()
}
```
⚙️ 8. Companion Objects and Apply Methods

A **companion object** shares the same name as a class and can access its private members.
```
class Config private(val url: String, val port: Int)
object Config {
  def apply(url: String, port: Int): Config = new Config(url, port)
}
object CompanionExample extends App {
  val conf = Config("localhost", 8080)
  println(s"Connected to ${conf.url}:${conf.port}")
}
```
🧠 9. Best Practices – Functional Programming for ETL

Principle	Description
Avoid mutable state	Makes code thread-safe for Spark parallelism

Use pure functions	Output depends only on input
Leverage immutability	Prevents side effects
Prefer map/filter/reduce	Declarative style over loops
Use Option/Try/Either	Handle missing or bad data gracefully

🗃️ 10. Working with Files (Scala I/O)

✅ Example: Reading/Writing Large Files

```
import scala.io.Source
import java.io._
object FileIOExample extends App {
  val source = Source.fromFile("input.csv")
  val lines = source.getLines().toList
  source.close()
  val writer = new PrintWriter(new File("output.csv"))
  lines.map(_.toUpperCase).foreach(writer.println)
  writer.close()
}
```

🧩 11. Integration with JSON, CSV, and Config Files

Using json4s for JSON

```
import org.json4s._
import org.json4s.jackson.JsonMethods._
object JsonExample extends App {
  implicit val formats = DefaultFormats
  val json = """{"name":"Dani","age":30}"""
  val parsed = parse(json)
  println((parsed \ "name").extract[String])
}
```

Using scala-csv for CSV

```
import com.github.tototoshi.csv._
object CSVExample extends App {
  val reader = CSVReader.open("data.csv")
  reader.allWithHeaders().foreach(println)
  reader.close()
}
```

⚙️ 12. Futures and Parallel Collections

🔹 Futures

Used for concurrent, non-blocking operations.

```
import scala.concurrent._
import ExecutionContext.Implicits.global
import scala.concurrent.duration._
object FutureExample extends App {
  val f1 = Future { Thread.sleep(1000); "Load Source" }
  val f2 = Future { Thread.sleep(500); "Transform Data" }
  val result = for {
    a <- f1
    b <- f2
  } yield s"$a and $b completed"
```

```
  println(Await.result(result, 3.seconds))
}
```

🔹 Parallel Collections

Used for easy data parallelism.

```
object ParallelExample extends App {
  val data = (1 to 10).toList.par
  val squares = data.map(x => x * x)
  println(squares)
}
```

🗒️ Summary Table

Topic	Key Takeaway	Common Use in ETL
Immutable vs Mutable	Safe vs performant	Spark transformations
HOFs	Functions as data	.map, .filter, .reduce
Currying	Partially applied config functions	ETL utilities
For-comprehensions	Clean chaining	Joining datasets
Lambdas	Inline transformations	Spark map/filter
Implicits	Auto configurations	Database connections
Classes/Traits	Reusability	Job abstraction
Companion Object	Factory pattern	Object creation
File I/O	Read/Write data	Raw file ingestion
JSON/CSV	Data format handling	Data ingestion pipelines
Futures/Parallel	Concurrency	Parallel ETL processing