Scala Language Fundamentals

Scala Language Fundamentals form the backbone for writing concise, expressive, and powerful Scala code. Here is a focused overview of the key fundamentals, with examples and explanations suited to Scala 2:

1. Basic Syntax and Data Types

- Variables
  - `val` defines immutable variables (cannot be reassigned).
  - `var` defines mutable variables (can be reassigned).

```
val name: String = "Scala"
var count: Int = 10
count = 15  // Allowed for var
```

- Common Data Types:
  - `Int`, `Double`, `Boolean`, `String`, `Unit` (like void in Java)

```
val age: Int = 30
val price: Double = 19.99
val isScalaFun: Boolean = true
def printHello(): Unit = println("Hello")
```

2. Control Structures

- If-Else

```
if (age > 18) println("Adult") else println("Minor")
```

- Match-Case (pattern matching)

```scala
val day = "Tuesday"
day match {
  case "Monday"    => println("Start of week")
  case "Friday"    => println("End of week")
  case _           => println("Midweek day")
}
```

- Loops
  - For loops: `for(i <- 1 to 5) println(i)`
  - While loops also supported but less used

3. Functions and Methods

- Define with `def`, specify input parameters and return type.
- Functions can be first-class values.

```scala
def square(x: Int): Int = x * x

val add = (a: Int, b: Int) => a + b  // Anonymous function stored in val
println(square(5))  // 25
println(add(3, 4))  // 7
```

4. Classes and Objects

- Classes define blueprints; objects are singletons.
- Constructor parameters can be class fields with `val` or `var`.

```scala
class Person(val name: String, var age: Int) {
  def greet(): String = s"Hi, my name is $name."
}

val p = new Person("Dani", 30)
println(p.greet())  // Hi, my name is Dani.
```

- Companion objects hold static-like members.

```scala
object Person {
  def species = "Homo sapiens"
}
println(Person.species)
```

5. Collections

- Collections are powerful and immutable by default.
- Common ones: `List`, `Array`, `Map`, `Set`

```scala
val nums = List(1, 2, 3, 4)
val doubled = nums.map(_ * 2)        // List(2, 4, 6, 8)
val evens = nums.filter(_ % 2 == 0) // List(2, 4)
```

6. Pattern Matching and Case Classes

- Pattern matching replaces switch/case statements with powerful matching capabilities.
- Case classes provide lightweight data structures with built-in equality and pattern matching support.

```scala
case class Person(name: String, age: Int)

def describePerson(p: Person): String = p match {
  case Person("Shivam", _) => "Shivam found!"
  case Person(name, age) if age < 18 => s"$name is a minor"
  case Person(name, age) => s"$name is $age years old"
}

val p = Person("Shivam", 25)
println(describePerson(p))
```

7. Immutability and Functional Style

- `val` and immutable collections enable safer, side-effect-free code.
- Use higher-order functions (`map`, `flatMap`, `filter`, etc.) to manipulate data.

8. Packaging and Imports

- Organize code with packages and import statements.

```
package com.example

import scala.collection.mutable.ListBuffer
```

9. Exception Handling

- Uses try-catch-finally blocks similar to Java.

```
try {
  val result = 10 / 0
} catch {
  case e: ArithmeticException => println("Cannot divide by
zero")
} finally {
  println("This always runs")
}
```

These fundamentals form the foundation of Scala programming, blending object-oriented and functional paradigms. With these concepts, you can write expressive and concise Scala applications and explore more advanced features like concurrency, implicits, and type classes as next steps.