SCALA PROGRAMMING MIND MAP (Complete)

```
Scala
|
├── 1. Basics
|      ├── Variables (val, var)
|      ├── Data Types
|      ├── Strings & String Interpolation
|      ├── Conditionals (if/else)
|      ├── Loops (for, while)
|
├── 2. Functions
|      ├── Defining functions
|      ├── Default & Named parameters
|      ├── Anonymous functions (lambdas)
|      ├── Higher-order functions
|      ├── Partially applied functions
|      ├── Function currying
|
├── 3. Collections
|      ├── Immutable collections
|      |        ├── List
|      |        ├── Seq
|      |        ├── Vector
|      |        ├── Map
|      |        ├── Set
|      ├── Mutable collections
|      |        ├── ArrayBuffer
|      |        ├── ListBuffer
|      ├── Transformations
|      |        ├── map
|      |        ├── filter
|      |        ├── flatMap
|      |        ├── fold / reduce
|      |        ├── groupBy
```

```
│
├── 4. Object-Oriented Concepts
│      ├── Classes & Objects
│      ├── Constructors
│      ├── Case Classes
│      ├── Traits
│      ├── Inheritance
│      ├── Companion Objects
│      ├── Abstract Classes
│
├── 5. Functional Programming
│      ├── Immutability
│      ├── Pure functions
│      ├── map, flatMap, filter pipeline
│      ├── Pattern Matching
│      ├── Partial Functions
│      ├── Higher-Kinded Types (Option, Either)
│      ├── Typeclasses (implicit behavior)
│
├── 6. Error Handling
│      ├── try/catch/finally
│      ├── Try, Success, Failure
│      ├── Option (Some/None)
│      ├── Either (Right/Left)
│
├── 7. Advanced Scala
│      ├── Implicits (implicit vals, defs)
│      ├── Extension Methods
│      ├── Pattern Matching Advanced
│      ├── Generics
│      ├── Enums (Scala 3)
│      ├── Recursion & Tail Recursion
│      ├── Collections Performance
│
├── 8. Scala + Spark Essentials
│      ├── SparkSession creation
│      ├── DataFrame → Dataset conversion (case classes)
│      ├── UDFs
│      ├── Joins, filters
│      ├── Window functions
│      ├── Broadcast joins
│
└── 9. Tools & Ecosystem
       ├── sbt
       ├── IntelliJ setup
       ├── REPL
       ├── File I/O
```

🧩 CODE SNIPPETS FOR EVERY TOPIC

1️⃣ SCALA BASICS

Variables

```
val name: String = "Dani"      // immutable
var age: Int = 30              // mutable
age = age + 1
```

Conditionals

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

Loops

```
for (i <- 1 to 5) println(i)
```

---

2 FUNCTIONS

Function definition

```
def add(a: Int, b: Int): Int = a + b
```

Anonymous function (lambda)

```
val square = (x: Int) => x * x
```

Higher-order functions

```
def operate(a: Int, b: Int, f: (Int, Int) => Int): Int = f(a, b)
operate(10, 20, _ + _)
```

Currying

```
def multiply(a: Int)(b: Int) = a * b
multiply(5)(2)
```

---

3 COLLECTIONS

Immutable List

```
val nums = List(1,2,3,4)
```

map, filter, reduce

```
nums.map(_ * 2)
nums.filter(_ % 2 == 0)
nums.reduce(_ + _)
```

groupBy

```
val grouped = List("apple", "ant", "bat").groupBy(_.charAt(0))
```

4️⃣ OOP

Class + Constructor

```
class Person(val name: String, val age: Int)

val p = new Person("Arun", 28)
```

Case class

```
case class Employee(id: Int, name: String)
```

Trait + Inheritance

```
trait Animal { def speak(): Unit }
class Dog extends Animal { def speak() = println("Bow Bow") }

new Dog().speak()
```

Companion object

```
class Counter(val value: Int)
object Counter {
  def apply(v: Int): Counter = new Counter(v)
}
val c = Counter(10)
```

5️⃣ FUNCTIONAL PROGRAMMING

Pattern matching

```
val x = 2
x match {
  case 1 => println("One")
  case 2 => println("Two")
  case _ => println("Other")
}
```

Option

```
val data: Option[String] = Some("Scala")
data.getOrElse("No value")
```

Either (Right = success)

```
def safeDivide(a: Int, b: Int): Either[String, Int] =
  if (b == 0) Left("Div by 0") else Right(a / b)
```

6️⃣ ERROR HANDLING

Try / Success / Failure

```
import scala.util.{Try, Success, Failure}

val result = Try(10 / 0)

result match {
  case Success(v) => println(v)
  case Failure(e) => println(e.getMessage)
}
```

---

7 ADVANCED SCALA

Implicits (auto behavior)

```
implicit val defaultMultiplier = 2

def multiply(x: Int)(implicit m: Int) = x * m

multiply(10)   // uses implicit 2
```

Generics

```
def first[A](list: List[A]): A = list.head
```

Tail recursion

```
@annotation.tailrec
def fact(n: Int, acc: Int = 1): Int =
  if (n <= 1) acc else fact(n - 1, n * acc)
```

---

8 SCALA + SPARK (CORE CONCEPTS)

SparkSession

```
val spark = SparkSession.builder()
  .appName("App")
  .master("local[*]")
  .getOrCreate()
```

DataFrame → Dataset

```
case class Person(id: Int, name: String)
val df = Seq((1,"a"), (2,"b")).toDF("id","name")
val ds = df.as[Person]
```

UDF

```
val toUpper = udf((s: String) => s.toUpperCase)
df.withColumn("upper", toUpper($"name"))
```

Window function

```
val w = Window.partitionBy("dept").orderBy($"salary".desc)
df.withColumn("rank", rank().over(w))
```

Broadcast join

```
large.join(broadcast(small), "id")
```

---

9 TOOLS

sbt build example

```
ThisBuild / scalaVersion := "2.12.18"

libraryDependencies ++= Seq(
  "org.apache.spark" %% "spark-core"  % "3.5.1",
  "org.apache.spark" %% "spark-sql"   % "3.5.1"
)
```
