

Scala Handson on Questions

Q1. Smart Temperature Converter (Conditionals & Basic Functions)

Write a program that defines a function:

```
def convertTemp(value: Double, scale: String): Double
```

- If scale is "C", convert Celsius to Fahrenheit.
- If "F", convert Fahrenheit to Celsius.
- Otherwise, return the input unchanged.

Example:

```
println(convertTemp(0, "C")) // 32.0
println(convertTemp(212, "F")) // 100.0
println(convertTemp(50, "X")) // 50.0
```

Learning focus:

- if / else if / else as **expressions** (not statements).
- Function return inference (= omitted vs included).
- Type safety on Double.

Q2. Array Mirror Puzzle (Arrays + Loops + Operators)

You're given an integer array. Create a **new array** that mirrors it.

Example:

Input: `Array(1, 2, 3)`
Output: `Array(1, 2, 3, 3, 2, 1)`

Write a function:

```
def mirrorArray(arr: Array[Int]): Array[Int]
```

Constraints:

- Use basic array operations only — no reverse or ++ directly.
- Use loops or comprehensions to construct the result.
- Must return a *new array* (immutability principle).

Learning focus:

- Working with arrays (indexing, length).
- Expression-style for loop in Scala (for ... yield).
- Immutable creation using yield.

Q3. Digit Sum using Simple Recursion

Write a **recursive** function to compute the sum of digits of a number.

Example:

```
digitSum(1345) => 13
```

Signature:

```
def digitSum(n: Int): Int
```

Hint:

- Base case: when $n == 0$, return 0.
- Recursive case: $(n \% 10) + \text{digitSum}(n / 10)$.

Learning focus:

- Pure recursion.
- Base and recursive cases.

- Integer arithmetic and remainder operator %.

Q4. Tail Recursive Factorial with Debug Prints

Implement a **tail-recursive** version of factorial that also prints how recursion unwinds.

Example:

```
factorial(5)
```

Should print something like:

```
[acc=1, n=5]
[acc=5, n=4]
[acc=20, n=3]
[acc=60, n=2]
[acc=120, n=1]
```

Signature:

```
def factorial(n: Int): Int
```

Requirements:

- Use an inner helper function annotated with @tailrec.
- Carry an accumulator parameter.
- Print at each step to observe tail recursion's nature.

Learning focus:

- Understanding accumulator usage.
- Seeing *how tail recursion replaces stack frames*.
- The @tailrec annotation benefit.

Q5. Find Max Element in an Array using Tail Recursion (No Loops)

Write a **tail-recursive** function:

```
def maxInArray(arr: Array[Int]): Int
```

Rules:

- Must not use loops or built-in max.
- Use a helper recursive function that moves through indices.
- Keep track of the current max as an accumulator.

Example:

```
val nums = Array(5, 9, 3, 7, 2)  
println(maxInArray(nums)) // Output: 9
```

Hint:

- You'll need a helper like def loop(i: Int, currentMax: Int): Int.
- Base case: $i == arr.length \rightarrow$ return currentMax.
- Recursive step: call with $i + 1$ and updated max.

Learning focus:

- Array traversal using recursion.
- Thinking in terms of **state carried through parameters**.
- Practical tail recursion pattern.

⚡ Scala For-Comprehension & Collections — Set 1 (Innovative + Educational)

Q1. Building a 2D Multiplication Table (Nested For-Comprehension)

Write a program that uses a **for comprehension** to build a **2D multiplication table** as a list of formatted strings.

Example:

```
val table = multiplicationTable(3)
table.foreach(println)
```

Expected output:

```
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
```

Signature:

```
def multiplicationTable(n: Int): List[String]
```

Learning focus:

- Nested generators in for.
- Producing flattened results (via automatic flatMap).
- String interpolation inside comprehensions.

Q2. Combining Two Lists Functionally (Cartesian Product + Filter)

You are given two lists:

```
val students = List("Asha", "Bala", "Chitra")
val subjects = List("Math", "Physics")
```

Write a for comprehension to generate all (**student, subject**) pairs,
but only include pairs where the student's name length is **greater than the subject's**.

Expected output:

```
List((Asha,Math), (Chitra,Math), (Chitra,Physics))
```

Learning focus:

- Combining multiple generators.
- Using if filters inside comprehensions.
- How for expands into flatMap + withFilter + map.

Q3. Extracting and Transforming Nested Data (flatMap Insight)

You are given:

```
val departments = List(  
    ("IT", List("Ravi", "Meena")),  
    ("HR", List("Anita")),  
    ("Finance", List("Vijay", "Kiran"))  
)
```

Write a for comprehension to create a **flat list** of formatted strings:

```
IT: Ravi  
IT: Meena  
HR: Anita  
Finance: Vijay  
Finance: Kiran
```

Hint: use pattern matching in generators like:

```
for {  
    (dept, employees) <- departments  
    emp <- employees  
} yield s"$dept: $emp"
```

Learning focus:

- Pattern decomposition in comprehensions.
- Implicit flattening via flatMap.
- Turning nested structures into a flat, human-readable list.

Q4. Functional Filtering and Transformation (Realistic Scenario)

You have a list of orders:

```
case class Order(id: Int, amount: Double, status: String)
val orders = List(
    Order(1, 1200.0, "Delivered"),
    Order(2, 250.5, "Pending"),
    Order(3, 980.0, "Delivered"),
    Order(4, 75.0, "Cancelled")
)
```

Use a for comprehension to extract only **delivered** orders whose **amount > 500**, and return a list of formatted strings:

```
Order #1 -> ₹1200.0
Order #3 -> ₹980.0
```

Learning focus:

- Real-world filtering.
- Working with case classes inside comprehensions.
- Expression-oriented list transformations.

Q5. Word Frequency Counter using Collections + For Comprehension

Given a list of sentences:

```
val lines = List(  
    "Scala is powerful",  
    "Scala is concise",  
    "Functional programming is powerful"  
)
```

Build a **word frequency map** ($\text{word} \rightarrow \text{count}$) using a for comprehension.

Expected output:

```
Map("Scala" -> 2, "is" -> 3, "powerful" -> 2, "concise" -> 1, "Functional" -> 1, "programming" -> 1)
```

Hint:

- Split each sentence into words.
- Flatten the result.
- Use groupBy + mapValues(_.size) after comprehension.

Learning focus:

- Real use of for comprehension for flattening nested results.
- Composition with collection methods (groupBy, mapValues).
- Transition from imperative text processing to functional pipelines.

These five together teach:

- How **for comprehensions** unify iteration, mapping, and flattening.
- How **functional collections** express real-world data flows concisely.
- How Scala avoids explicit loops while keeping logic readable.

Set: Scala Operators — Hands-On Learning

Q1. The Mystery of `:+` and `:+`:

You're given a list:

```
val nums = List(2, 4, 6)
```

Write Scala expressions (not functions) to produce the following results **using only `:+` and `+:` operators**:

1. Add 8 **to the end** of the list.
2. Add 0 **to the beginning** of the list.
3. Build a new list [0, 2, 4, 6, 8] using these operators.

 *Learning hint:*

Understand that `:+` appends at the end, `+:` prepends at the start.

Try to **chain** them to build new sequences immutably.

Q2. Operator Overload — Reverse Thinking

Define a class:

```
class Counter(val value: Int)
```

Implement a custom `+` operator so that:

```
val a = new Counter(5)
val b = new Counter(7)
println(a + b) // should print 12
```

and an overloaded version where:

```
println(a + 10) // should print 15
```

 *Learning hint:*

In Scala, you can define:

```
def +(that: Counter): Int = ...
def +(that: Int): Int = ...
```

See how **methods can act like operators** seamlessly.

Q3. The Curious Case of :: and List Construction

Without using List() directly, build the list [1, 2, 3, 4] using only the :: and Nil operators.

Example:

```
val myList = ??? // fill this to create List(1, 2, 3, 4)
```

 *Learning hint:*

:: is **right-associative** — so parentheses are not where you expect them.

See how Scala builds lists from the right!

Q4. ++ vs ::— The Merge Game

You're given:

```
val a = List(1, 2)  
val b = List(3, 4)
```

Try both:

```
val c1 = a ++ b  
val c2 = a :: b
```

Print both results and note the difference.

Then, using your observation, combine two lists **and** a vector to form a single collection of all elements.

 *Learning hint:*

Both ++ and :: are used for **collection concatenation**, but they behave differently depending on collection types.

Q5. The -> Arrow Puzzle

You're building a map of animals to their sounds:

```
val animals = Map(  
    "dog" -> "bark",  
    "cat" -> "meow",  
    "cow" -> "moo"  
)
```

Now:

1. Add "lion" -> "roar" using the same syntax.
2. Retrieve the sound of "cow".
3. Try to access "tiger" safely using getOrElse.



-> is just syntactic sugar for creating a pair (key, value).

scala Functional Concepts

Q1. Mood Transformer (Higher-Order Function + Closure)

Design a function generator that creates a **mood changer**.

```
def moodChanger(prefix: String): String => String = {  
    word => s"$prefix-$word-$prefix"  
}
```

Now write your own def moodChanger so that:

```
val happyMood = moodChanger("happy")  
println(happyMood("day")) // "happy-day-happy"
```

```
val angryMood = moodChanger("angry")
println(angryMood("crowd")) // "angry-crowd-angry"
```

 *Learning aim:*

You're creating a **function that returns another function** — one that *remembers* a value (prefix).

This demonstrates *closures* — inner functions capturing outer scope variables.

Q2. Sentence Pipeline (Function Composition with Transformation Chain)

Given:

```
val trimSpaces: String => String = _.trim
val toLower: String => String = _.toLowerCase
val capitalizeFirst: String => String = s => s.head.toUpperCase + s.tail
```

Compose them into a single processSentence function that:

1. Removes extra spaces.
2. Converts to lowercase.
3. Capitalizes the first letter.

Example:

```
val messy = " HeLLo WOrld "
println(processSentence(messy)) // "Hello world"
```

 *Learning aim:*

Instead of data flow, focus on **behavior flow** — chaining transformations declaratively.

Q3. Delayed Greeter (Partial Application + Time Control)

Create a function:

```
def delayedMessage(delayMs: Int)(message: String): Unit = {
    Thread.sleep(delayMs)
    println(message)
}
```

Now:

1. Create a **partially applied** function that always delays by 1000ms:

```
val oneSecondSay = delayedMessage(1000)_
```

2. Use it to print multiple lines with one-second gaps.



Functions can “store” timing behavior — partial application here binds *how* a function behaves (its timing), not *what* it prints.

Q4. Adaptive Discount (Higher-Order Function Returning Strategy)

Write a function:

```
def discountStrategy(memberType: String): Double => Double
```

It should return a *different* discount function:

- "gold" → 20% off
- "silver" → 10% off
- otherwise → no discount

Usage:

```
val goldDiscount = discountStrategy("gold")
println(goldDiscount(1000)) // 800.0
```



You're generating *custom logic* at runtime — a **strategy pattern** purely through functional programming.

Q5. Intentional Crasher (Partial Function with Safety Net)

Define a **PartialFunction[Int, String]** called safeDivide such that:

- For numbers other than zero → "Result: " + (100 / x)
- For zero → *not defined*

Now, handle it safely:

```
val safe = safeDivide.lift  
println(safe(10)) // Some("Result: 10")  
println(safe(0)) // None
```



You experience the power of partial functions as **controlled failure zones** — only *defined where valid*.

By lifting, you transform *danger* into *Option safety*.

❖ Set: Currying · Pattern Matching · Option · Either — Functional Mindset

Q1. The Personalized Calculator (Currying + Partial Application)

Define a **curried function** that takes two steps:

```
def calculate(op: String)(x: Int, y: Int): Int
```

It should perform:

- "add" → $x + y$
- "sub" → $x - y$
- "mul" → $x * y$
- "div" → x / y

Example:

```
val add = calculate("add")_
val multiply = calculate("mul")_
println(add(10, 5)) // 15
println(multiply(3, 4)) // 12
```

 *Learning aim:*

Currying turns a multi-argument function into **stages of intent** — here, you first pick *operation*, then *operands*.

This mimics *configurable logic pipelines*.

Q2. The Shape Analyzer (Pattern Matching on Case Classes)

Define two case classes:

```
case class Circle(r: Double)
case class Rectangle(w: Double, h: Double)
```

Write a function:

```
def area(shape: Any): Double = shape match {
  // handle Circle
  // handle Rectangle
  // default: return -1.0
}
```

Example:

```
println(area(Circle(3)))    // 28.27...
println(area(Rectangle(4, 5))) // 20.0
```

 *Learning aim:*

Pattern matching is *destructuring + decision logic* in one shot — this encourages modeling data as **typed cases** instead of ad hoc conditionals.

Q3. The Safe Divider (Option)

Write a function:

```
def safeDivide(x: Int, y: Int): Option[Int]
```

- If $y == 0$, return None
- Else return Some(x / y)

Then:

```
val result = safeDivide(10, 0).getOrElse(-1)
println(result) // -1
```

 *Learning aim:*

You experience **Option as a wrapper for uncertainty** — not exception-based error handling, but **semantic safety**.

Q4. The Login Validator (Either for Error Handling)

Write:

```
def validateLogin(username: String, password: String): Either[String, String]
```

- If username is empty → Left("Username missing")
- If password is empty → Left("Password missing")

- Else → Right("Login successful")

Example:

```
println(validateLogin("", "123")) // Left.Username missing
println(validateLogin("user", "")) // Left.Password missing
println(validateLogin("user", "123")) // Right.Login successful
```



Learning aim:

Either expresses **failure or success as values**, not exceptions.

Use Left for problems, Right for success.

Q5. The Smart Parser (Pattern Matching on Option & Either)

Combine your previous logic:

You get a user input number as String.

Write:

```
def parseAndDivide(input: String): Either[String, Int]
```

Behavior:

1. Try to convert input to Int (use `toIntOption`).
 - a. If conversion fails → Left("Invalid number")
2. Divide 100 by the number using `safeDivide` from Q3.
 - a. If divide fails → Left("Division by zero")
 - b. Else → Right(result)

Then:

```
println(parseAndDivide("25")) // Right(4)
println(parseAndDivide("0")) // Left(Division by zero)
println(parseAndDivide("abc")) // Left(Invalid number)
```

 *Learning aim:*

Pattern matching can **unbox nested logic elegantly**, composing Option and Either into a clean, declarative workflow — no try/catch, no nulls, no confusion.

Set: Advanced Scala — Apply/Unapply · Code Blocks · Lazy Evaluation

Q1. The apply Evaluator — Code Block as Input

Design an object Evaluator that takes a **code block** as input to its apply method, executes it, and prints both:

1. the expression itself (as a string)
2. the evaluated result

Example:

```
object Evaluator {  
    def apply(block: => Any): Unit = {  
        println(s"Evaluating block...")  
        val result = block  
        println(s"Result = $result")  
    }  
}
```

```
Evaluator {  
    val x = 5  
    val y = 3  
    x * y + 2  
}
```

Expected output:

Evaluating block...

Result = 17

 *Learning aim:*

You're passing a **code block as a by-name parameter**, deferring its execution until apply decides.

This is a foundational Scala pattern — powering DSLs and frameworks like Play or Akka.

Q2. The Dual Nature Object — Apply & Unapply Mirror

Create an object Email that can **both construct and deconstruct** email addresses:

```
object Email {  
    def apply(user: String, domain: String): String =  
        s"$user@$domain"  
  
    def unapply(email: String): Option[(String, String)] = {  
        val parts = email.split("@")  
        if (parts.length == 2) Some((parts(0), parts(1))) else None  
    }  
}
```

Now:

```
val e = Email("alice", "mail.com")  
println(e) // alice@mail.com
```

```
e match {  
    case Email(user, domain) => println(s"User: $user, Domain: $domain")  
    case _ => println("Invalid email")  
}
```

 *Learning aim:*

apply builds, unapply extracts — a *true bidirectional interface*.

You now treat an *object as a constructor and a pattern simultaneously* — very “Scala”.

Q3. Lazy Evaluation with Triggers

Create a class LazyCounter that counts how many times its value was *actually computed*, even when accessed multiple times.

```
class LazyCounter {  
    private var computeCount = 0  
    lazy val value: Int = {  
        computeCount += 1  
        println("Computing value...")  
        42  
    }  
    def getCount: Int = computeCount  
}
```

Now:

```
val c = new LazyCounter  
println("Before first access")  
println(c.value)  
println("Access again")  
println(c.value)  
println("Compute count: " + c.getCount)
```

 *Expected result:*

Before first access

Computing value...

42

Access again

42

Compute count: 1

Learning aim:

You see **memoization in action** — lazy evaluates once, then caches.
This teaches the difference between *lazy val*, *def*, and *val* in practical terms.

Q4. Unapply Chain — Nested Pattern Matching

You're given:

```
case class Address(city: String, pincode: Int)
case class Person(name: String, address: Address)
```

Write a pattern match that extracts the city and pincode *directly* from a Person object in a single case statement.

```
val p = Person("Ravi", Address("Chennai", 600001))
```

```
p match {
  case Person(_, Address(city, pin)) =>
    println(s"$city - $pin")
  case _ => println("No match")
}
```

Now modify it so that you only print if the city starts with "C", using a **pattern guard** (if condition).

 *Learning aim:*

Deep destructuring + pattern guards = **concise logic trees** — one of Scala's most elegant control-flow mechanisms.

Q5. Lazy Pipeline Builder (Lazy Evaluation + Apply Block + Closure Retention)

Design a Pipeline object that builds **deferred computation pipelines** using code blocks.

```
object Pipeline {  
    def apply[T](block: => T): LazyPipeline[T] = new LazyPipeline(block)  
}  
  
class LazyPipeline[T](block: => T) {  
    lazy val result = block  
    def map[R](f: T => R): LazyPipeline[R] = Pipeline(f(result))  
}
```

Usage:

```
val p = Pipeline {  
    println("Step 1: Preparing data")  
    List(1, 2, 3)  
}.map { xs =>  
    println("Step 2: Transforming data")  
    xs.map(_ * 2)  
}  
  
println("Before accessing pipeline...")  
println("Result: " + p.result)
```

 *Expected output:*

```
Before accessing pipeline..  
Step 1: Preparing data  
Step 2: Transforming data  
Result: List(2, 4, 6)
```

Set: Operator Overloading & Implicits — Innovative Hands-On Questions

Q1. Create a Vector Algebra DSL using Operator Overloading

Design a Vec2D class representing 2D vectors that supports natural syntax for arithmetic:

```
val v1 = Vec2D(2, 3)
val v2 = Vec2D(4, 1)
```

```
println(v1 + v2)  // Vec2D(6,4)
println(v1 - v2)  // Vec2D(-2,2)
println(v1 * 3)   // Vec2D(6,9)
println(3 * v1)   // Vec2D(6,9)
```

Requirements:

1. Overload +, -, and * operators.
2. Use an **implicit conversion** so `3 * v1` works (integer on the left).
3. Override `toString` for neat output.

 *Learning aim:*

Use **operator methods** (`def +(that: Vec2D)`) and **implicit conversions** to simulate a *mini algebraic DSL* — intuitive and mathematical.

Q2. Implicit Ordering with Custom Operator

Define a class `Person(name: String, age: Int)` and an **implicit ordering** so that you can compare two persons directly with `>`, `<`, `>=`, `<=` based on age.

Example:

```
val p1 = Person("Ravi", 25)
val p2 = Person("Meena", 30)
```

```
println(p1 < p2) // true  
println(p1 >= p2) // false
```

Requirements:

- Use **implicit class** to add the comparison operators to Person.
- Comparison should be based on age only.
- Ensure expressions like if (p1 > p2) ... work naturally.



Showcases how implicits **retrofit existing types with operators**, and how Scala lets you make your own “operator universe”.

Q3. String Enrichment via Implicit Class + Operator

You want to build a small DSL where you can “repeat” strings using * and “concatenate with spacing” using ~.

Example:

```
println("Hi" * 3) // "HiHiHi"  
println("Hello" ~ "World") // "Hello World"
```

Requirements:

- Use an **implicit class RichString** to add both operators.
- Ensure normal string behavior remains intact.



Learn how **implicit enrichment** works — it’s not inheritance, it’s *type extension by scope* (a cornerstone of Scala’s expressivity).

Q4. Implicit Parameter for Precision in Operator

Create a Money class that supports addition and subtraction, but automatically rounds to the nearest multiple of a given precision (say ₹0.05, ₹0.10, etc.).

Example:

```
implicit val roundingPrecision: Double = 0.05
```

```
val m1 = Money(10.23)
val m2 = Money(5.19)
println(m1 + m2) // Money(15.20)
```

Requirements:

- Use an **implicit parameter** for precision.
- Implement + and - using rounding rules.
- Make rounding customizable via implicit values in scope.



Shows how **implicit parameters** let you tune global behavior without passing config everywhere — powerful for DSLs and APIs.

Q5. Chained Implicit Conversions — Building an Int to Rational Bridge

You're building a small rational-number class:

```
case class Rational(n: Int, d: Int)
```

You should be able to write:

```
val r = 1 / Rational(2, 3) // means Rational(3,2)
println(r)               // Rational(3,2)
```

Requirements:

1. Use an **implicit conversion** from Int to Rational.
2. Overload / inside Rational for division between rationals.
3. Show that normal Int / Int still works without ambiguity.

Set: Exception Handling + Function Composition + Collections Operations

Q1. Safe Division — Functional Exception Handling

Write a function:

```
def safeDivide(a: Int, b: Int): Either[String, Double]
```

It should return:

- Right(result) if division is valid.
- Left("Division by zero") if b == 0.

Then, using a list of pairs:

```
val pairs = List((10, 2), (5, 0), (8, 4))
```

map it using safeDivide, collect only the valid results, and print:

Valid results: List(5.0, 2.0)

Errors: List("Division by zero")



Move away from try/catch — model errors as *values* using Either.

This is *pure functional error handling* that's safer and composable.

Q2. Function Pipelines — Compose vs andThen

You have three functions:

```
val trim: String => String = _.trim  
valToInt: String => Int = _.toInt  
val doubleInt: Int => Int = _ * 2
```

1. Compose them using both compose and andThen to build a pipeline:
 - a. "21" → 42
2. Show how swapping the order in compose changes evaluation flow.



Hands-on understanding of **function composition direction**:

- f compose g ⇒ f(g(x))
- f andThen g ⇒ g(f(x))

This distinction becomes vital when chaining transformations.

Q3. Collections + Partial Functions — Selective Transformations

Given a list of mixed elements:

```
val items = List(1, "apple", 3.5, "banana", 42)
```

Use a **partial function** and collect to:

- Extract only integers and double them.
Expected output:

```
List(2, 84)
```



- Understand how **collect + partial function** unifies *filter + map*.

- Explore the power of *pattern matching* inside collections.

Q4. Retry Logic — Functional Exception Recovery

Define a function:

```
def fetchData(): Int = {
  val n = scala.util.Random.nextInt(3)
  if (n == 0) throw new RuntimeException("Network fail")
  n
}
```

Now:

1. Write a **retry mechanism** `retry(times: Int)(op: => T): Option[T]` that tries up to times and returns `Some(result)` or `None` if all fail.
2. Use it as:

```
val data = retry(3)(fetchData())
println(data)
```



Learning aim:

- Use **higher-order functions** with **by-name parameters** ($=> T$).
- Functionalize exception handling: no mutable retry counters, no catch clutter.

Q5. Map–FlatMap Composition — Clean Data Pipeline

You have:

```
val data = List("10", "20", "x", "30")
```

Write a pipeline that:

1. Tries to convert each string to an Int safely (using Try).
2. Filters out failed conversions.
3. Squares all valid numbers.
4. Returns List(100, 400, 900).

You may chain map, flatMap, or collect to do this *without any explicit ifs or try-catch*.

 *Learning aim:*

- Learn how map and flatMap interact with **monadic containers** (Try, Option, List).
- Think of control flow as *data transformation*, not branching.

Set: Futures & Asynchronous Programming — Hands-On Questions

Q1. ⏱ Future Sequencing — Parallel vs Sequential

You have three simulated tasks:

```
import scala.concurrent._  
import scala.concurrent.ExecutionContext.Implicits.global  
import scala.concurrent.duration._  
import scala.util._  
  
def task(name: String, delay: Int): Future[String] = Future {  
    Thread.sleep(delay)  
    s"$name done"  
}
```

1. Call them **sequentially** (using flatMap).
2. Then call them **in parallel** (using Future.sequence).
3. Measure and print the total time taken for both approaches.

 *Learning aim:*

Understand how **Future runs eagerly** and how Future.sequence composes multiple

async tasks *without blocking*.

Learn to distinguish *sequential dependency* from *parallel independence*.

Q2. Safe Composition — Future with Recover and RecoverWith

You have a function that randomly fails:

```
def riskyOperation(): Future[Int] = Future {  
    val n = scala.util.Random.nextInt(3)  
    if (n == 0) throw new RuntimeException("Failed!")  
    n  
}
```

Tasks:

1. Call this operation and print either success or fallback value -1 using recover.
2. Then rewrite using recoverWith to *retry* once (return a new Future instead of a fixed value).



Learn the subtle but crucial difference between:

- recover: returns *a fallback value*
- recoverWith: returns *a fallback future (retry)*
This models *resilient async control flow*.

Q3. Chaining Dependent Futures — Function Composition

Suppose you have:

```
def getUser(id: Int): Future[String] = Future { s"User$id" }  
def getOrders(user: String): Future[List[String]] = Future { List(s"$user-order1", s"$user-order2") }  
def getOrderTotal(order: String): Future[Double] = Future
```

```
{ scala.util.Random.between(10.0, 100.0) }
```

Build a pipeline that:

1. Fetches user 42
2. Gets all their orders
3. Computes and prints the **total amount**

Use flatMap chaining (or for-comprehension).



Learning aim:

Grasp **function composition in the async world** — each step depends on the previous, but nothing blocks.

Use *for-comprehension* to flatten nested futures elegantly.

Q4. Combine Futures with Dependencies — MapN Style

You have three independent futures:

```
val f1 = Future { Thread.sleep(1000); 10 }
val f2 = Future { Thread.sleep(800); 20 }
val f3 = Future { Thread.sleep(500); 30 }
```

Create a new future that produces:

"Sum = 60, Average = 20"

without blocking or awaiting individual results manually.

You must use only **combinators (map, flatMap, zip, sequence, etc.)**, not Await.result.



Learning aim:
Understand **zip and mapN-style** future composition — combining independent async results cleanly and functionally.

Q5. Coordinated Retry — Asynchronous Resilience

Implement:

```
def fetchDataFromServer(server: String): Future[String]
```

which randomly fails (50% chance).

Now write:

```
def fetchWithRetry(server: String, maxRetries: Int): Future[String]
```

that retries automatically up to maxRetries on failure.

Usage:

```
fetchWithRetry("Server-1", 3).onComplete(println)
```



Learning aim:

Master **recursive retry** with Futures — requires understanding of *flatMap*, *recoverWith*, and *non-blocking recursion*.

Shows how to implement retry logic *without loops or blocking*, using pure async recursion.