# Scala Interview Handbook — Batch 1

Generated: 2025-09-13 02:45:31Z (UTC)

## Scala Theory & Cheatsheet

```
SCALA THEORY & CHEATSHEET (Quick Ref)
-------------------------------------
- Syntax: vals vs vars; methods; case classes; pattern matching.
- Collections: immutable List/Vector/Map/Set; mutable variants.
- Functional: map/flatMap/filter/fold; Options/Eithers; for-comprehensions.
- Performance: prefer immutable + structural sharing; use arrays for tight loops.
- Testing: ScalaTest AnyFunSuite; assertions; property-based (optional).
```

# 001. ReverseString

## Problem Overview & Strategy

ReverseString — Detailed Explanation Approach: Use hash maps / frequency arrays and linear scans. Correctness: Frequencies capture necessary character counts; single pass maintains invariants. Complexity: O(n) time, O(Σ) space.

## Scala Solution

```
package problems

object Problem001ReverseString {
  def reverseString(s: String): String = if (s == null) null else s.reverse
}
```

## ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem001ReverseString

class Problem001ReverseStringSpec extends AnyFunSuite {
  test("basic") {
    assert(Problem001ReverseString.reverseString("abc") === "cba")
    assert(Problem001ReverseString.reverseString("") === "")
    assert(Problem001ReverseString.reverseString(null) == null)
  }
}
```

# 002. IsPalindrome

## Problem Overview & Strategy

IsPalindrome — Detailed Explanation Approach: Expand-around-center for every index (odd/even centers). Correctness: The longest palindrome must expand from some center; scanning all centers guarantees finding it. Complexity: O(n^2) time, O(1) space. Edge cases: empty string, all identical chars, multiple optimal answers.

## Scala Solution

```
package problems

object Problem002IsPalindrome {
  def isPalindrome(s: String): Boolean = {
    if (s == null) false else s == s.reverse
  }
}
```

## ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem002IsPalindrome

class Problem002IsPalindromeSpec extends AnyFunSuite {
  test("examples") {
    assert(Problem002IsPalindrome.isPalindrome("racecar"))
    assert(!Problem002IsPalindrome.isPalindrome("hello"))
  }
}
```

# 003. CharFrequency

## Problem Overview & Strategy

CharFrequency — Detailed Explanation Approach: Idiomatic Scala solution with clear invariants and tested behavior. Correctness: Proven by invariant reasoning and unit tests. Complexity: See code comments.

## Scala Solution

```
package problems

object Problem003CharFrequency {
  def charFrequency(s: String): Map[Char, Int] =
    Option(s).map(_.toList.groupBy(identity).view.mapValues(_.size).toMap).getOrElse(Map.empty)
}
```

## ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem003CharFrequency

class Problem003CharFrequencySpec extends AnyFunSuite {
  test("freq") {
    val m = Problem003CharFrequency.charFrequency("aab")
    assert(m('a') === 2)
    assert(m('b') === 1)
  }
}
```

# 004. FirstNonRepeatedChar

## Problem Overview & Strategy

FirstNonRepeatedChar — Detailed Explanation Approach: Idiomatic Scala solution with clear invariants and tested behavior. Correctness: Proven by invariant reasoning and unit tests. Complexity: See code comments.

## Scala Solution

```
package problems
object Problem004FirstNonRepeatedChar {
  def firstNonRepeated(s:String): Option[Char] = {
    if (s==null) None
    else {
      val counts = s.foldLeft(scala.collection.mutable.LinkedHashMap.empty[Char,Int])((m,c) => { m(c)=m.getOrElse(c,0)+
      counts.collectFirst{ case (ch,c) if c==1 => ch }
    }
  }
}
```

## ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem004FirstNonRepeatedChar

class Problem004FirstNonRepeatedCharSpec extends AnyFunSuite {
  test("first non-repeated") {
    assert(Problem004FirstNonRepeatedChar.firstNonRepeated("aabbc").contains('c'))
  }
}
```

# 005. SecondLargest

## Problem Overview & Strategy

SecondLargest — Detailed Explanation Approach: Idiomatic Scala solution with clear invariants and tested behavior. Correctness: Proven by invariant reasoning and unit tests. Complexity: See code comments.

## Scala Solution

```
package problems
object Problem005SecondLargest {
  def secondLargest(a:Array[Int]): Option[Int] = {
    if (a==null || a.length<2) None
    else {
      var first: java.lang.Integer = null
      var second: java.lang.Integer = null
      for (n <- a) {
        if (first==null || n>first) { second=first; first=n }
        else if (n!=first && (second==null || n>second)) second=n
      }
      Option(second).map(_.intValue)
    }
  }
}
```

## ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem005SecondLargest

class Problem005SecondLargestSpec extends AnyFunSuite {
  test("second largest") {
    assert(Problem005SecondLargest.secondLargest(Array(1,3,2)).contains(2))
  }
}
```

# 006. RemoveDuplicates

## Problem Overview & Strategy

RemoveDuplicates — Detailed Explanation Approach: Idiomatic Scala solution with clear invariants and tested behavior. Correctness: Proven by invariant reasoning and unit tests. Complexity: See code comments.

## Scala Solution

```
package problems
object Problem006RemoveDuplicates {
  def dedup(a:Array[Int]): Array[Int] = if (a==null) null else a.distinct
}
```

## ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem006RemoveDuplicates

class Problem006RemoveDuplicatesSpec extends AnyFunSuite {
  test("dedup") {
    assert(Problem006RemoveDuplicates.dedup(Array(1,2,1)).sameElements(Array(1,2)))
  }
}
```

# 007. RotateListK

## Problem Overview & Strategy

RotateListK — Detailed Explanation Approach: Bounds pointers (top/bottom/left/right) for spiral; transpose+reverse for rotation; classic i-k-j loops for multiplication. Correctness: Each layer/element is moved/visited exactly once. Complexity: O(mn) time.

## Scala Solution

```
package problems
object Problem007RotateListK {
  def rotateRight(a:Array[Int], k:Int): Array[Int] = {
    if (a==null || a.isEmpty) a
    else {
      val n=a.length; val kk=((k % n)+n)%n
      (a.takeRight(kk) ++ a.dropRight(kk))
    }
  }
}
```

## ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem007RotateListK

class Problem007RotateListKSpec extends AnyFunSuite {
  test("rotate") {
    assert(Problem007RotateListK.rotateRight(Array(1,2,3,4),1).sameElements(Array(4,1,2,3)))
  }
}
```

# 008. MergeTwoSortedLists

## Problem Overview & Strategy

MergeTwoSortedLists — Detailed Explanation Approach: Use appropriate sorting—Merge/Quick/Heap—or selection (min-heap/quickselect). Correctness: Follows standard algorithms with proven invariants. Complexity: typically O(n log n) sort; quickselect average O(n).

## Scala Solution

```
package problems

object Problem008MergeTwoSortedLists {
  final class ListNode(var x:Int, var next: ListNode)
  object ListNode { def apply(x:Int): ListNode = new ListNode(x, null) }

  def merge(a: ListNode, b: ListNode): ListNode = {
    val dummy = ListNode(0)
    var t = dummy; var p=a; var q=b
    while (p!=null && q!=null) {
      if (p.x < q.x) { t.next = p; p = p.next }
      else { t.next = q; q = q.next }
      t = t.next
    }
    t.next = if (p!=null) p else q
    dummy.next
  }
}
```

## ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem008MergeTwoSortedLists

class Problem008MergeTwoSortedListsSpec extends AnyFunSuite {
  test("merge two sorted lists") {
    val A = Problem008MergeTwoSortedLists.ListNode(1); A.next = Problem008MergeTwoSortedLists.ListNode(3)
    val B = Problem008MergeTwoSortedLists.ListNode(2); B.next = Problem008MergeTwoSortedLists.ListNode(4)
    val R = Problem008MergeTwoSortedLists.merge(A,B)
    assert(R.x == 1 && R.next.x == 2)
  }
}
```

## 009. LinkedListCycle

## Problem Overview & Strategy

LinkedListCycle — Detailed Explanation Approach: BFS/DFS for traversal & cycle detection; Kahn for topological order; Dijkstra with a min-heap. Correctness: Standard graph invariants (visited sets, indegrees, relaxation) guarantee optimality. Complexity: O(V+E) for BFS/DFS/Topo; O((V+E) log V) for Dijkstra.

## Scala Solution

```
package problems

object Problem009LinkedListCycle {
  final class Node(var v:Int, var next: Node)
  object Node { def apply(v:Int): Node = new Node(v, null) }

  def hasCycle(h: Node): Boolean = {
    var slow = h; var fast = h
    while (fast!=null && fast.next!=null) {
      slow = slow.next; fast = fast.next.next
      if (slow eq fast) return true
    }
    false
  }
}
```

## ScalaTest

```
package tests
import org.scalatest.funsuite.AnyFunSuite
import problems.Problem009LinkedListCycle

class Problem009LinkedListCycleSpec extends AnyFunSuite {
  test("cycle") {
    val a = Problem009LinkedListCycle.Node(1); val b = Problem009LinkedListCycle.Node(2); val c = Problem009LinkedListC
    a.next=b; b.next=c; c.next=b
    assert(Problem009LinkedListCycle.hasCycle(a))
  }
}
```

# 010. DetectCycleLinkedList

## Problem Overview & Strategy

DetectCycleLinkedList — Detailed Explanation Approach: BFS/DFS for traversal & cycle detection; Kahn for topological order; Dijkstra with a min-heap. Correctness: Standard graph invariants (visited sets, indegrees, relaxation) guarantee optimality. Complexity: O(V+E) for BFS/DFS/Topo; O((V+E) log V) for Dijkstra.

## Scala Solution

```
package problems
object Problem010DetectCycleLinkedList {
  final class Node(var v:Int, var next:Node)
  object Node { def apply(v:Int)= new Node(v,null) }
  def hasCycle(h:Node): Boolean = {
    var s=h; var f=h
    while (f!=null && f.next!=null) {
      s=s.next; f=f.next.next
      if (s eq f) return true
    }
    false
  }
}
```

## ScalaTest

```
package tests

import org.scalatest.funsuite.AnyFunSuite
import problems.Problem010DetectCycleLinkedList

class Problem010DetectCycleLinkedListSpec extends AnyFunSuite {
  test("cycle") {
    val a=Problem010DetectCycleLinkedList.Node(1); val b=Problem010DetectCycleLinkedList.Node(2); val c=Problem010Detec
    assert(Problem010DetectCycleLinkedList.hasCycle(a))
  }
}
```