

Scala Training



Course Available

An in-house introductory [Scala training course](#) is offered to new hires and is available for individuals and teams coming to Scala for the first time. It's a hybrid of self-paced with a live workshop to kick off, and is run frequently. You are recommended to take the course, and use this page and its exercises as a supplement.

- [Quick Start](#)
- [Beyond the Basics](#)
- [Deep References](#)
- [The Ten Commandments](#)
 - 1. Never use "null" (use Option instead)
 - 2. Avoid 'for' loops, use map instead
 - 3. Case classes should be used as data containers and only as data containers
 - 4. Avoid "var" as much as possible (use "val" instead)
 - 5. Prefer immutable types over mutable types
 - 6. Avoid explicit casting and type checking, use pattern matching instead
 - 7. Companion objects are should be used if they are beneficial to the class they are companioned with
 - 8. Data dependencies should be resolved with laziness
 - 9. Implicit arguments should be of specialised types
 - 10. It's better to use classes than objects
- [Testing in Scala](#)
 - [Mocking](#)
- [Configuration in Scala](#)
- [Scala Exercises](#)
- [Concurrency in Scala](#)
- [Git Training](#)
- [Hadoop & Spark Training](#)
 - [Hadoop book](#)
 - [Spark](#)
 - [Exercises](#)
- [Related articles](#)

Quick Start

This easy-to-follow video introduces Scala with a walkthrough of features.

This "cheat-sheet" style presentation gives examples of language features and constructs.



scala_workshop.pptx



Beyond the Basics

Once you have a feel for how OOP and imperative features you're used to from other languages work in Scala (you've reached the "Java Without Semicolons" level), [The Neophyte's Guide to Scala](#) series of articles is an excellent next step. These cover many concepts that most people will not have encountered before Scala, but will quickly become things you use every day. You'll likely refer back to this series many times in your first weeks, months, and even years with the language.

Read Parts 1-9 in full, and the rest as your time and interest allows.

Deep References

For comprehensive reference, the following books are recommended reading.

Scala / Functional Programming	Scala Testing
Programming_Scala_Second_Edition.pdf <i>This edition is outdated and lacks coverage of Scala 2.12;</i> <i>Essential Scala</i> is a good alternative that is more compact, free, and more current.	 Testing_i...Scala.pdf
Everything up to (and including) Chapter 12, but not Chapter 6 pages 199-206 and pages 227-242, the whole Chapter 7 pages 243-266, Chapter 8 pages 281-294. (Total of around 308 pages)	Read chapters 1-3 "ScalaMock" under chapter 5
<div> Source Code The source code for examples in book can be found in <code>git clone git@github.com:deanwampler/prog-scala-2nd-ed-code-examples.git</code></div>	

The Ten Commandments

1. Never use "null" (use Option instead)

```
def findById(id: Int): Option[Record]
```

Null pointers are the source of a lot of evil stuff in computer programming, in lesser languages we're often forced to write code littered with **IF** conditions to make sure that we don't end up invoking methods on a null reference somewhere far away from where the null first snuck into the system. Benefits of not using null are:

- You'll avoid null pointer exceptions (NPEs).
- No spaghetti **if-then-else** code to check null references.
- The type signature tells you when a method may not always have a value to return.
Declaring a method to return an `Option[T]` type is a terrific way to indicate to the caller that they need to expect a possible "no result" case, `None`, in addition to a present/happy path value (a `Some[T]`). This is a *much* better approach than returning `null` from a method that claims it returns an object type. This is like taking good care of your customers (and it leads to good business 😊). Consumers of your code will know in advance (*at compile time*) to be prepared and most importantly that your API will not break their code with a null reference exception.
- More expressive code.
In addition to types giving static information about their possible absence, using Options and their combinator methods like `map`, `flatMap`, `filter`, and `exists` leads you to a more declarative style of programming. There's a transformative effect that helps you learn a safer, functional style that you will see again and again.
- Interoperation with the standard library. The Scala collection libraries have better support for Options and they expect you to use Options if you have uninitialised values.

In Scala every declaration (**val/var/def**) needs to be initialized immediately so controlling this temptation of initializing with `null` becomes all the more important.

Null does not exist in *idiomatic* Scala; if not for Scala's deep Java interoperability we might be able to forget it exists at all. If a Scala library—third-party or internal—consider it buggy and file an issue. When you need to write Scala code interacting with Java APIs, wrap potential null cases to prevent them leaking out to the rest of your program.

These articles are helpful to reinforce the **Why** and **How** about not using `null`:

[Scala best practices: null values, Option, Some, and None](#)

[Scala best practice: Eliminate null values from your code](#)

2. Avoid 'for' loops, use map instead

```
val words = List("foo", "bar")

// DO NOT
val buf = ListBuffer.empty[String]
for (word <- words) {
  buf += word.toUpperCase
}
val upcased = buf.result

// DO
val upcased = words.map(word => word.toUpperCase)
```

Not only less code, but also more declarative rather than imperative—emphasis is on *what* is being done instead of *how* a loop works.

In Scala the common case is transforming an immutable collection, not updating a mutable one. This ensures thread safety and paves the way to parallel operations (maybe you've heard of map/reduce...).

3. Case classes should be used as data containers and only as data containers

4. Avoid "var" as much as possible (use "val" instead)

5. Prefer immutable types over mutable types

6. Avoid explicit casting and type checking, use pattern matching instead

7. Companion objects are should be used if they are beneficial to the class they are companioned with

8. Data dependencies should be resolved with laziness

9. Implicit arguments should be of specialised types

10. It's better to use classes than objects

The Scala `object` keyword defines singleton objects. These are good for *static* data and methods, like constants and pure utility functions. Don't use them for state—using `objects` for mutable state will lead to the bad, troublesome design patterns characteristic of globals, to concurrency synchronization bugs, and to code that is difficult if not impossible to unit test properly.

Testing in Scala

The book *Testing in Scala*, although now dated, still covers in detail the essentials of the main test frameworks in the Scala ecosystem: `ScalaTest`, `specs2`, and `ScalaCheck`. All are used in Agoda in different projects—find out what your team uses and focus on reading about it.

[Get a totally legit copy here](#) or on O'Reilly Safari (that's `\\bk-agfil-1001\IT\eBooks\Testing_in_Scala.pdf` if your inferior OS won't open the link).

- Read Chapters 1 and 2
- Chapter 3 for `ScalaTest`
- Chapter 4 for `specs2`

Mocking

Most projects in Agoda use either `ScalaMock` or `Mockito` for mocking. Again, learn what your team uses and:

- Read Chapter 5

If you're starting new projects, `ScalaMock` is recommended because it supports language features of Scala that `Mockito` (a Java framework) cannot handle well.

Configuration in Scala

Read the README for <https://github.com/lightbend/config> up until (not including) "Miscellaneous Notes".

Lightbend/Typesafe Config is widely used throughout the Scala ecosystem—it's likely to be used by libraries and frameworks you'll work with, such as Akka and Play.

Your team may use [PureConfig](#) or [Ficus](#) as Scala wrappers for Typesafe Config, so you may want to learn them, but you need to know Typesafe Config first in either case.

Scala Exercises

All exercises should have tests!

Pick the [ScalaTest style](#) your team prefers for unit tests, or [specs2 unit specification style](#) if your team uses specs2.

Please review all exercises with your dev manager or the mentor they suggest

1. Use a trait to define a generic queue of strings with 'put' and 'get' methods, and create a class that implements it using an array. Include tests.
2. Starting from the previous exercise, use a [stackable trait](#) to modify the behavior of 'put' so it reverses each string (e.g. hello to olleh) before adding it to the queue. Include tests.
3. Write the `~=` operator for comparing doubles. The operator should return true iff 2 doubles are equal up to a small constant. The constant should be configurable but also have a default. (hint: use implicit parameter) Include tests.
4. Implement GCD in Scala (hint: use pattern matching and tail recursion)
5. Write a clause that measures the run time of a block of code and prints it (also needs to return the original output of the block):

```
timeit {  
  ...  
  ...  
}
```

6. Given a string containing words separated by space, find:
 - a. The longest word
 - b. The most common word
 - c. The most common letter
 - d. Create a map from letter to a set of words it appears in
7. Convert a list of strings to a list of all the characters in all the strings
8. Given the following code:

```
trait IntSet {  
  def incl(x: Int): IntSet  
  def contains(x: Int): Boolean  
}  
class EmptySet extends IntSet {  
  def contains(x: Int): Boolean = false  
  def incl(x: Int): IntSet = new NonEmptySet(x, new EmptySet, new EmptySet)  
}  
class NonEmptySet(elem: Int, left: IntSet, right: IntSet) extends IntSet {  
  def contains(x: Int): Boolean =  
    if (x < elem) left contains x  
    else if (x > elem) right contains x  
    else true  
  def incl(x: Int): IntSet =  
    if (x < elem) new NonEmptySet(elem, left incl x, right)  
    else if (x > elem) new NonEmptySet(elem, left, right incl x)  
    else this  
}
```

Write methods union and intersection to form the union and intersection between two sets.

```
trait IntSet {  
  ...  
  def union(x: IntSet): IntSet  
  def intersect(x: IntSet): IntSet  
}
```

Add a method

```
def excl(x: Int)
```

to return the given set without the element x. To accomplish this, it is useful to also implement a test method

```
def isEmpty: Boolean
```

for sets.

9. Consider the following definitions representing trees of integers (Binary Search Tree). These definitions can be seen as an alternative representation of IntSet:

```
abstract class IntTree
case object EmptyTree extends IntTree
case class Node(elem: Int, left: IntTree, right: IntTree) extends IntTree
```

Complete the following implementations of function contains and insert for IntTree's.

```
def contains(t: IntTree, v: Int): Boolean = t match { ...
    ...
}
def insert(t: IntTree, v: Int): IntTree = t match { ...
    ...
}
```

10. Consider a function which squares all elements of a list and returns a list with the results. Complete the following two equivalent definitions of squareList.

```
def squareList(xs: List[Int]): List[Int] = xs match {
    case List() => ??
    case y :: ys => ??
}
def squareList(xs: List[Int]): List[Int] = {
    xs map ??
}
```

11. Write the following functions:
- A function that gets optional x, y, and z and returns the first that is not None
 - A function that gets optional Ints x, y, and z and if all are defined (Some(...)), returns their product.
 - A function that gets a sequence of optional elements and returns the first that is not None.
12. Given a list: List[Int] and map: Map[Int, Double], multiply all the numbers in the list with their corresponding value in the map, and drop if don't exists. For example, list = [1,2,3,4], map = {1 -> 3, 3-> 5} ==> res = [3, 15]
13. Write a retry method that converts a method to a retry-able method.

The syntax should look like:

```
retry { ... }
```

However, you also need some way to specify how many times to retry and support sleep between retries (hint: implicit arguments)

14. Design a class that is given a vector of numbers in the constructor and exposes:
- x: a vector with the square of all elements in the input vector
 - y: the sum of x
 - z: the square root of y
- Nothing should be calculated in the constructor of the class assume the calculation of x, y, z can take a lot of time, and should only be done once (at most)
15. Add a method "median" to a Seq of integers so that s.median is the media of s for s of type Seq[Int]
How can you add the same method for a sequence of doubles with minimal code duplication?
16. Basic scalalab exercises <https://github.com/scala-labs/scala-labs/tree/master/labs/src/test/scala/org/scalalabs/basic>

Concurrency in Scala

Our [intro training course](#) and this page mostly cover sequential programming. That's because concurrency is a large topic, and there are many options available in Scala—your learning path from here will branch depending on what you need in your team: Futures; Akka Actors; Rx and Reactive Streams like Akka's, Monix, FS2; Cats Effect, etc. Ask your teammates for suggestions.

[Learning Concurrent Programming in Scala](#) is a solid reference that surveys many of these paradigms, written by an authoritative source: one of the authors of standard library Futures and parallel collections. The book should be available on O'Reilly Safari, ask your manager about how to access it. There are a few paper copies available on BKK 22, bookshelf beside the pantry.

If you need to work with Futures (unless most of your development is on Spark, you probably will), these are helpful places to start:

- [The Neophyte's Guide to Scala](#), Parts 8 and 9
- The *Futures in Scala Protips* series on [Viktor Klang's blog](#)

Git Training

Make sure you are familiar with all the things written here:

- <https://www.atlassian.com/git/tutorials/viewing-old-commits/>
- <https://www.atlassian.com/git/tutorials/undoing-changes>
- <https://www.atlassian.com/git/tutorials/syncing>
- <https://www.atlassian.com/git/tutorials/rewriting-history>
- <https://www.atlassian.com/git/tutorials/using-branches>
- <https://www.atlassian.com/git/tutorials/merging-vs-rebasing>
- <https://www.atlassian.com/git/tutorials/resetting-checking-out-and-reverting/>

Hadoop & Spark Training

If your team uses Spark, a formal in-house training course is available—ask in [#spark-training](#) for current details. For self-study see below.

Hadoop book

Book - [Hadoop: The Definitive Guide](#) (\\bk-agfil-1001\IT\eBooks\Hadoop_The_Definitive_Guide_Third_Edition.pdf)

- HDFS - Chapter 3 (Up to command line interfaces - inclusive)
- Hive - chapter 12 (from "comparison with traditional databases" to "querying data" - inclusive) - (Sirilak is our expert on this subject)
- OOOIE
 - <http://hortonworks.com/hadoop/oozie/>
 - <http://www.infoq.com/articles/oozieexample>

Install VirtualBox and Vagrant. Install the ML vagrant machine (anyone in the team can assist with this).

Spark

Book - [Learning Spark](#) (\\bk-agfil-1001\IT\eBooks\Learning_Spark.pdf)

- Chapter 2 - from *Introduction to Core Spark Concepts* (page 14) to the end of the chapter
- Chapters 3,4,9
- Chapter 7 - from *Introduction to Executors*
- Clone this project on Spark Scala Training and go through its readme: <https://github.com/deanwampler/spark-scala-tutorial>

See [the Machine Learning group's main Spark page](#) to find more resources and Agoda custom tooling for Spark like app deployment and testing helpers.

Exercises

- Try [the Machine Learning group's Spark Exercises](#).
- Clone this project and go through its readme: <https://github.com/deanwampler/spark-scala-tutorial>

Related articles

- [Scala Training](#)
- [Continuous Integration for Scala project in TeamCity](#)
- [Code Coverage Scala / Teamcity](#)

Related issues
