About This Guide

YoppWorks created The Beginner's Guide to Learning Scala to help people like you. People who are interested in learning how to code in Scala. We understand that learning a new language can be difficult, and sometimes overwhelming. This step-by-step guide will walk you through learning Scala from scratch on your own. We've pulled together the best, most trusted resources from industry experts on each topic to make sure that you're spending your time efficiently and learning Scala the right way.

Quizzes Each section of the guide is also equipped with a short quiz, allowing you to test your skills on the topic before moving on. Don't worry, this is not a real test! The quizzes are simply there for your benefit. If you get all of the answers right, you're probably ready to move on to the next topic. If you struggle, go back and review the learning resources again.

How To Use The Guide This guide will help you learn Scala at your own pace. Here are some quick tips for using this guide: • Track your progress by checking off the topics as you go. • Mark topics for later by favouriting them using the star icon. • Take advantage of the My Notes section to leave yourself tips or thoughts for later. • Join in the Public Discussion to ask questions, leave feedback or provide tips to others who are trying to learn Scala. Before we get started, let's quickly review why you might want to learn Scala.

Why Scala?

If you're here, you're probably convinced you want to try your hands at Scala. But it doesn't hurt to summarize the main reasons Scala can be a good fit for a number of applications:

- Maintainability: Scala allows developers to express a lot with little code. This power, when used wisely, results in code that is faster to comprehend and easier to maintain (since there is less reading and visual parsing to do).
- Scala speaks JVM: There is a huge amount of good libraries written in Java, which you can use from your Scala code seamlessly. Don't reinvent the wheel if you don't need to.
- Object Oriented AND Functional: Both paradigms have strengths, and you're free to combine them with Scala. Create classes, objects, higher order functions, and powerful type hierarchies - there's plenty of power in the language.
- Strong ecosystem: The frameworks, tools and libraries around Scala are growing stronger by the minute. Play, Akka, Spray, and many others are already being used in very large systems, and adoption continues to accelerate.
- Ready to get started with Scala? The first step is setting up your environment. Click the "next" button below to begin!

Setting Up Your Environment

As with any new language, you'll need to first setup your development environment.

Follow the simple steps in this section to get started!

Install Scala

Before installing Scala, you want to be sure to have Java's JDK 1.8 installed on your machine. For any operating system, you can also download the binaries from the Scala language website and add the Scala interpreter and the Scala compiler to your path as explained on Scala-Lang.org.

Both Mac and Linux operating systems offer installation methods using package managers.

On Mac OSX

The simplest way to install Scala is using a package utility such as Homebrew or MacPorts.

On Linux

The simplest way to install Scala on Linux is via the package manager.

Under Ubuntu or Debian, this will be:

```
sudo apt-get install scala
```

Under Fedora, this will be:

```
sudo yum install scala
```

Other Linux distributions may have a different or no package manager at all, in which case you will likely need to install Scala manually. These instructions can help you accomplish this.

Choosing An IDE

The two most popular Scala IDE's are Eclipse and IntelliJ. There are a variety of other IDE's which support Scala, such as vim, Sublime, and emacs, but for the purposes of this guide we'll focus on these two.

Setting up IntelliJ

By default, IntelliJ is set for Scala projects to be compiled externally. To compile through the IDE, you will need to point to the compiler when you create the project.

Setting up Eclipse

When it comes to Eclipse, you have two options: download the core Eclipse IDE and configure extensions manually, or download Scala IDE for Eclipse. Scala IDE for Eclipse is the more convenient of the options, but the stable version often works with an earlier version of Eclipse.

Now you're ready to move on to **Hello World** and make your first Scala app!

Scala Build Tool (Optional)

You're almost done setting up your environment - there's one more optional tool.

The **Scala Build Tool**, or **sbt**, is an open-source tool to assist in the compilation of Scala applications. It is similar to Java tools such as Maven or Ant. It is very popular in the Scala community and is used by multiple frameworks.

While it is optional, we recommend you look at installing it and familiarizing yourself with using it for compilation. The best way to install it on your platform can be found at the scala-sbt.org site.

Now that you're all set up, it's time to move on the the basics of Scala!

The Basics

This section will give you a tour of the fundamentals of the language - the main building blocks you'll need to understand and write Scala code.

We assume you have an understanding of some programming language. If you come from a Java or C# background, many Object Oriented constructs will feel familiar. If you speak Haskell, you'll recognize the Functional aspects of the language.

Walk through the fundamentals and take the time to understand each element of the language. Be sure to take the short quiz at thee end of each section to ensure that you've understood the concept before moving on.

If you're not into the "Do It Yourself" approach to learning Scala, and want to get ramped up and productive fast, check out our Fast Track to Scala training course. This two day course is taught by Typesafe certified trainers and is available virtually, meaning that you can attend from anywhere in the world.

Starting Fundamentals

Hello World

Scala-lang.org, the curators of Scala, have a good article guiding you through a basic Hello World application. They cover coding directly in the interpreter and how to begin scripting in the language. Go check it out, we'll wait right here...

Feel free to try the example on the REPL or from your favourite IDE.

Classes & Objects

Let's get started with Classes and Objects. Classes are the fundamental templates we use to create objects in memory which reflect the business glossary of the problem we're trying to solve. This article at Scala-lang.org demonstrates many features of Scala classes.

In addition, singleton objects in Scala implement a pattern that is very common in many languages. Singleton objects are created without a class declaration. Examples can be found here.

Ready to move on? Test your knowledge on Classes and Objects.

Test

```
1. Which fields are publicly available on instances of the following class?

class Notebook(pages: Int, val colour: Colour)
{       val sheets = pages / 2
        private var content: String  }

 A. pages, colour, sheets and content
 B. pages, colour and sheets
 C. sheets
 D. colour and sheets
 E. None, this class has no public fields

Answer:
D:Colour and Sheets

2.What method is called when an object is called directly? (e.g. Class(parameters) )

A. The class constructor
B. The companion object's apply method
C. None of the above

Answer:
B: The companion object’s apply method.

3.How do you create a static class method?

A. Use the "static" keyword before the method
B. Create a "static class" with the same name
C. You don't. Create a companion object with the same name and put the method definition there.
D. You don't. All methods are static in Scala.

Answer:
D: You Don’t. Create a companion object with the same name and put the method definition there.

4.How do you access the constructor of a singleton object?

A. By using the new keyword and the object name
B. You don't. Singleton objects are instances of their own special class, which is not available
from Scala code.
C. Through the apply method

Answer:
B: You don’t.  Singleton objects are instances of their own special class, which is not
available from Scala code.
```

Fields

Class instances keep state in their fields, which can be immutable or mutable. Immutable data is always encouraged in Scala, but there are scenarios where you absolutely need mutation.

Class fields can be explicitly or implicitly typed. In the latter case, the compiler infers the type of the field based on the value assigned to it.

Furthermore, the Scala compiler generates getters and setters (when necessary) for fields, which reduces the amount of code we need to write and maintain.

Read Understanding Classes and its Fields in Scala by Mohamed Sanualla to learn more about fields.

Test your knowledge below to make sure you understand Fields.

```
1. Which members get generated by the scala compiler for the title field in the following class?

class Book(var title: String)

A. A public field
B. A private field and two methods: a getter and a setter
C. A private field and a getter method
D. Two methods: a getter and a setter

Answer:
C: A private field and twomethods: a getter and a setter.

Can the type of a field be omitted?
A. Yes, the compiler can infer the type of a field in all cases
B. No, it has to be explicitly typed
C. Only if the field is initialized with a value
D. Only if the field is a class parameter

Answer:
A:Only if the field is initialized with a value.

#### Case Classes
```

Case classes are a special type of classes, that offer a number of features in addition to regular classes. To illustrate case classes, we will walk through this modeling exercise:

Product-category model

- A product that has a name, category, and an optional description
- A category that has a name and may have a parent category
- All of the above attributes are read-only, which means that they are immutable.

In plain-old java, we could model the product-category class like this:

```
        public class Category{

        private String name;
        private Category parent;

        public Category(String name, Category parent)
        {
            this.name = name;
            this.parent = parent
            }

        public String getName(){
            return parent;
            }
      }

    public class Product{
        private String name;
        private String description;
        private Category category;

        public Category(String name, String description, Category category){
            this.name = name;
            this.description = description;
            this.category = category;
            }

        public String getName(){
            return name;
            }

        public String getDescription(){
            return descripion;
            }

        public Category getCategory(){
            return category;
            }
      }
```

And how do you write this in scala?

Simply as follows:

```
class Category(val name: String, val parent: Option[Category])

class Product(val name: String, val description: String, val category: Category)
```

The category class has the classical pattern. This you will find also in scala, C# and other object-oriented languages in which you have the class definition, your private variable(s) declaration(s), public constructor(s) and the get method(s) to create instances.

As you may have observed, we write fewer lines of codes in scala. This is one big reason, you should be using scala.

Or even better, you can create the case classes as follows:

```
case class Category(name: String, parent: Option[Category])

case class Product(name: String, description: String, category: Category)
```

and, you get the following benefits:

- Companion object with factory-like apply methods
- Class parameters become fields
- Nicer toString
- Attribute based equality
- Pattern matching
- Copy method to 'clone' instances

but, you have to be mindful of issues such as bytecode overhead due to the large code size of extra methods generated, and the generated code might clutter your interface, as well as you cannot inherit from or extend case classes based on the concept of inheritance.

And you can learn even more by reading this post by Daniel Spiewak in which he explains what case classes are, how to use them, and what makes them so useful.

Think you've got a handle on Case Classes? Prove it!

```
1. Which of the following do you get for free by using a case class instead of a regular class?
    A. Copy function
    B. Useful toString function
    C. Class parameters as fields
    D. Value-based equality checking
    E. Easy pattern matching over the class
    F. Companion object with factory method
    G. All of the above

    Answer:
    G:All of the Above

2. Is it possible to create a case class with mutable fields/parameters?
    A. Yes, all fields in a case class are mutable
    B. Yes, using the "var" keyword before the parameter
    C. No, instances of a case class are always immutable
    D. Only by creating mutable fields in the body of the class

    Answer:
    B: Yes, using the "var" keyword before the parameter

3. Which of the following does not return an instance of Ring, if the class is defined like
this: case class Ring(size: Int)
    A. new Ring(1)
    B. Ring(2)
    C. Ring.apply(5)
    D. Ring.unapply(new Ring(6))
    E. (new Ring(3)).copy()
    F. (new Ring(4)).copy(size = 2)

    Answer:
    D:Ring.unapply(new Ring(6))
```

Methods And Functions

Classes contain methods, and functions are first class citizens of the language. Sometimes we use these terms interchangeably, but they're strictly different.

To understand their relationship, read this article by Jim McBeath.

If you want more depth, these two articles can help:

- Learning Scala Part 5 - Methods - by Joel Abrahamsson
- Functional Scala: Turning Methods into Functions - by Mario Gleichmann

Have a good understanding of methods and functions? Take the short quiz below to make sure before you move on to Expressions vs. Statements.

```
    1. Which of the following are valid identifiers in Scala?
        A. myFunction
        B. myFunction+
        C. +
        D. function89097
        E. All of the above
        F. None of the above

        Answer: A

    2. When the code value1 + value2 is called, what is happening?
        A. The global "+" function is being called with value1 and value2 as parameters
        B. value1 is an object with a "+" method, called with value2 as a parameter
        C. value2 is an object with a "+" method, called with value1 as a parameter
        D. Operators are hard-coded into Scala and can not be changed


    3. Which of the following is false in Scala?
        A. All values are objects
        B. All values are functions
        C. All functions are translated into objects at runtime
        D. All functions have an apply method
```

Expressions And Statements

Scala is an expression-oriented language, which has implications on flow control and structure of some language constructs.

Got it? Take a crack at the questions below to make sure.

```
1. The "if else" construct in Scala is an example of a(n)
    A. Expression
    B. Statement
    C. None of the above

2.If an expression only returns Unit, can it be considered a statement?
    A. Yes, because the value it returns is meaningless
    B. No, because it still returns a value

3. Can you assign the result of a statement to a variable?
    A. Depends on the position of the statement
    B. Yes
    C. No
```

Object Oriented And Functional

Object Oriented is a powerful paradigm that models concepts as hierarchies of classes. Objects are instances of classes, which contain fields for state and methods for behaviour.

Functional Programming proposes other principles, such as pure functions and functions as first class citizens.

To understand how this works and the potential benefits, check out the following resources:

- Scala as a Functional OO Hybrid - by Toby Weston
- Why the debate on Object-Oriented vs. Functional Programming is all about composition - by Erkki Lindpere

Test your knowledge of this subject before moving on to the next section.

```
1. What defines a pure function?
    A. It does nothing except for calling other functions
    B. It takes no parameters
    C. There are no side-effects, i.e. nothing outside of the function is changed
    D. There is no return value, i.e. a value of Unit is returned
```

(Im)mutability

Immutability refers to data that does not change - a value is set once and never modified. Conversely, mutability refers to data structures that do change through time.

Scala encourages immutability over mutability. This can prevent a number of issues regarding concurrent reads and writes of shared data.

Another good reason why your data structure should be immutable is because you can keep control of your states so you don't get unexpect values. Moreover, most container types are immutable by default(collections for example).

The following points explains the concept of immutability, why it is important, how we can use it in Scala.

Create immutable variable in scala

Immutable variables are created with *val*. For example:

```
    val temperature = BigInt(-10)    // Creates an immutable variable

    temperature = 5                  // This gives a compilation error: cannot reassign to val
```

And to create mutable variable use *var*.

```
    var temperature = BigInt(-10)    // Creates an immutable variable

    temperature = 5                 // reassigned with no compilation error but  not recommended
```

Don't modify, create

However, you can create a new instance of the immutable variable that does not modify the original reference. For example:

```
    val absolute  = temperature.abs    // temperature is not changed and absolute gets BigInt
i.e. absolute = 10
```

Use copy

Case classes gives you a copy method, and immutabilty makes this easy to use as we see in the following example:

```
    case class User(name: String, email: String)  //Case class User has immutable String
attributes: name and email

    /* create new copies of the same class */
    val bilbo = User("Biblo", "bilbo@shi.re")
    val newBilbo = bilbo.copy(email = "bilbo@vali.or")

    /* let's print the output */
    println(bilbo)
    :User(Biblo,bilbo@shi.re)

    println(newBilbo)
    :User(Biblo,bilbo@vali.or) // Done! copied. Notice that  name remains the same while only
email is changed

    /* use copy method to change the name and email */
    val newerBilbo = bilbo.copy(name ="Moris", email = "bilbo@gra.mi") /* name and email is
changed
    :User(Moris,bilbo@gra.mi)
```

Hide your var

with regular class, it is vital to hide your variables which is *public* by default by making them *private*. The following explains this:

```
    class User(name: String){

      private var status = Status.Initial  //Public by default, hide it!

      /* use some funtions that are public by defaults to change the value */
      def transition(s:Status){
      status = s
    }
```

The idea is to be able to control as much as possible your mutable state.

You have to be mindful of the following when exploring (im)mutability in Scala.

- Modifying immutable nest structures i.e. having many copies
- Fields that get initialized with each new instance
- Mutable state that is not labeled as private
- Closing over mustable state

Don't forget to take the short quiz below on (Im)mutability before moving on to Inheritance!

```
1. When using the immutable "val" keyword to define a variable containing a Java array, can the
contents of this variable be mutated?
A. No
B. Yes, but only the contents - the array can not be replaced
C. Yes, everything can be changed

2. Immutability can make a system easier to understand because:
A. Outer scopes cannot access immutable members
B. It removes the possibility of multiple components modifying state.
C. Values are assigned at object construction time
D. Values are assigned when they are first used
```

Inheritance

Inheritance is the fundamental concept required to create hierarchies of classes to represent concepts and sub-concepts. This is used extensively in the Object Oriented camp, so learning the rules and conventions of inheritance is fundamental for a Scala developer.

Ted Neward explains inheritance in detail in this article.

Make sure you understand this concept before moving on - take the quiz below!

```
1. Can a class override a final member from one of its ancestors?
    A. Only if the overridden member is abstract
    B. No, final members cannot be overridden
    C. Yes, all members can be overridden

2. What is preventing this code snippet from compiling?

    class Person { def age: Int }

    A. The "case" modifier is missing before the "class" keyword
    B. The method definition should be on a new line
    C. The class should be abstract, since it has one abstract member
    D. Classes must have at least one class parameter

3. What members does a class inherit from its superclasses?
    A. Only fields
    B. Only methods
    C. All non-private fields and methods
    D. All non-final fields and methods
```

Traits

Traits offer a convenient way to modularize functionality, and to combine multiple components. They are an intermediate between Java's interfaces and abstract classes (in fact, the Scala compiler generates interfaces and classes for each trait).

This article by Chris Hodapp provides a great overview of traits in Scala.

Test your knowledge on Traits before advancing to Pattern Matching.

```
1. Can a class extend multiple classes?
    A. Yes
    B. No, but it can mix in multiple traits
    C. No, you can only extend one class or trait
    D. You can extend one concrete class and many abstract classes

2. Is it possible for a trait to mix in one or more traits?
    A. Yes, one or more traits
    B. A trait can only extend one trait
    C. No, traits can not extend anything

3. What is wrong with this trait definition:
   A. trait One Dimensional(length: Int)?
   B. Trait names should start with lowercase
   C. The body of the trait is missing
   D. Traits cannot have parameters
   E. Traits must have at least one member
```

Pattern Matching

Pattern matching is an extremely useful feature of the language, and can provide a more elegant replacement for class casting, and switch statements.

To illustrate pattern matching in Scala, we start with the following example:

- Users have accounts
- Accounts are either Paypal or Bitcoin
- Papal accounts have an email
- Bitcoin accounts have key

We model above example as folows:

```
case class User(name: String, account: String)
sealed Account
case class Paypal(email: String) extends Account
case class Bitcoin(key: String) extends Account
```

You may be familiar with the Java implementation of pattern matching which is can be done with if-else statements as we can illustate for the example we have:

```
private String whatis(Object obj){
    if (obj instanceof User){
     User user = (User)obj;
      if(user.account instanceof Paypal){
       Paypal paypal = (Paypal)user.account;
       return "Paypal for email" + paypal.email;
      } else if(user.account instanceof Bitcoin){
       Bitcoin bitcoin = (Bitcoin)user.account;
       return "Bitcoin with key" + bitcoin.key;
     }else
     {
       return "Unknown account type"
     }else{
       return "Unknown object type"
     }
  }
}
```

In Scala, pattern matching is more concise as follows:

```
def whatis(obj: Any): String = x match{
obj match{
    case User(name, Paypal(email)) =>      //match user name with Paypal email
    "Paypal for email" + email

    case User(name, Bitcoin(key))  =>     //match user name with Bitcoin key
    "Bitcoin with key" + key

    case_=>"Unknown object type"          //Default case

  }
}
```

Next, read Martin Odersky's in depth introduction to Pattern Matching .

You should be well-versed with Pattern Matching now - double check, just to make sure you've got it

```
1.Which of the following are true about pattern matching?
    A. Case classes can easily be used on constructor patterns to decompose objects into their
parts
    B. Patterns can be combined to form more complex patterns
    C. A pattern matching expression always returns a value
    D. A non-exhaustive pattern match can generate a MatchError if a value does not match any of
its cases
    E. All of the above

2. Why is the following pattern matching problematic?

    someValue match { case list:List[Int] => list.head }

    A. There is no break statement
    B. Type erasure prevents the runtime to determine parameterized types
    C. There is no default case

3. Other than match expressions, where can patterns be used to decompose objects?
    A. Variable definitions
    B. For expression generators
    C. For expression definitions
    D. try-catch expressions
    E. All of the above
```

Higher Order Functions

One of the main features that functional languages offer is the ability to treat functions as objects - pass them around and assign them to variables.

Functions that take other functions as parameters or that return other functions as results are called Higher Order Functions. This ability is extremely powerful, and is showcased very well in the collections API, and several other types in the Scala core library.

Set

In the following example we will show how to use Set Higher Order functions in Scala.

```
    Example: Filter

    Set[A] contains a method called filter:
        filter(p:A => Boolean): Set[A]
```

What this means is there is a collection api that contains Set[A], which are collections of things that are not repeated. And, in Set[A] you have a function called filter that allows you to pass in another function called predicate, p, that goes from A to a Boolean. Basically, you want a function that you can try on each element of set[A] that gives you a true/false; and that p will return you yet another collection of similar type, Set[A].

In order words, you want to be able to select elements of Set[A] that comply with a predicate. Let us illustrate further with our Filter example:

```
    val set = (-10 to 10).toSet     //convert value set to a Set using toSet method
    def isEven(i:Int) = i%2 == 0    // define a function that check if a number is even

    //Set[A].filter(p: A => Boolean):Set[A]
    // Here are three ways to  go about doing the filter.

    set.filter(i => isEven(i))
    set.filter(isEven(_))
    set.filter(isEven)
```

Eitherway you choose to use the filter as shown above, you will get the same result: Set(0, 10, -8, 6, -4, 2, -10, 8, -6, 4, -2). Scala gives you options of how to pass a function into another order function and as you go deeper into learning Higher Order Functions you will understand the advantages and drawbacks of doing this. But, for now, let's keep it simple.

Map

Another example of Higher Order functions is a Map that can be defined as follows:

```
Example: map
List[A] contains a method called map:
map[B](f:A => B): List[B]
```

Map is used to transform each element in a List[A] by applying function f in order to get another List[B] or simply to transform the elements in a collection. The following example will make this clearer:

```
def capitalize(s: String) =
s.head.toUpper + s.tail.toLowerCase

val list = List("samewise", "meriadoc", "peregrin")

list.map(capitalize)
```

so, what happens in the above map example is List("samewise", "meriadoc", "peregrin") is tranformed to List(Samewise, Meriadoc, Peregrin) using the capitalize function. And, of course everything is immutable - if you have already read our discussions on im(mutability).

Few things to bear in mind using Higher Other Functions are:

- Allow behaviour to be passed in an arguments
- Return functions from method calls
- Allows for more abstract constructs => more library-like, requiring less maintenance

Other interesting examples:

- Collections
    - best implementations depends on collection: recursive, iterative, with auxiliary mutable state
- Parallel collections
    - interace is fundamentally the same, parallization is hidden from you

- Futures
    - transform the value once it is available

Another great resource for learning about higher order functions is Mario Gleichmann's post Functional Scala: High, Higher, Higher Order Functions

You've read the post - now take the quiz!

```
1. In collections, the map method takes a function f as a parameter. What does f receive as a
parameter?
    A.The collection
    B. One item of the collection at a time
    C. Only the first item of the collection
    D. Nothing; it contains a reference to the collection through a closure

2. In collections, filter is a method that takes a function p as a parameter. What does p return
as a result?
    A. An item from the collection, wrapped in an Option
    B. An item from the collection, or null
    C. True or false

3. Given the following two methods, which of the following compiles?
    A. def test(f: User => Account) = ...
    B. def getAccount(u: User): Account = ...
    C. test(user:User => getAccount(user))
    D. test(user => getAccount(user))
    E. test(getAccount(_))
    F. test(getAccount)
    G. All of the above
```

Collections

The collections library in Scala is very powerful, and exemplifies several basic and advanced features of the language. It contains the usual types, such as Set, List, Array, and Map, but you'll also find more specialized ones like Stream, BitSet and Range.

Essentially, what you have in the collections api are:

- Very rich library for manipulating collection objects
- Great examples of higher order functions
- The usual suspects: Lists, Sets, Hash Maps
- but also Streams, Buffers, Arrays, Trees

So we move on to exploring the main features of Scala collections, and point to some simple examples.

First, let's take our simple product-category model:

```
case class Category(name: String, parent: Option[Category])

case class Product(name: String, description: String, category: Category)
```

Given this model, some of what you can do include:

- Sort them by name
- Get their names in uppercase
- Get the set of categories(no repeats)
- Group them by the initials of their names
- Return a tuple with the product and category names

Sorting

We can very simply use SortBy method in the list, To illustrate 'Sorting Product by name' we have:

```
val products = List(...)  //List ofproducts

products.sortBy(_.name)  //sort by name
```

Uppercase

You can get the names of the lists of product in uppercase using map:

```
val names = List[String] =  products.map(_.name.toUpperCase)    //Get their names in uppercase
```

Extracting

You can get the set of categories(no repeating) using map toSet method to do the transformation:

```
val catNames: Set[String] = products.map(_.category.name).toSet  //Get category name set
```

Grouping

Say with the lists of product we want to group them by the initials of their names such that you can easily navigate to where your product is using keys, in this case the initials of the names taken as chars. The code looks like this:

```
val grouped: Map[Char, List[Product]] = products.groupBy(_name.head)
```

So, what this code does is that the groupBy method takes a function that calculates the key you want to go by such that it extracts the name of the product taken the head as only the first character. Hence, this gives an hashmap that is keyed to the return type of Char in our case and points to a list of products which you can easily indexed.

More mapping

A last example we can get from collection is to create tuples of products of product and category names. This you use along with for expressions that are look a lot like for loops but are implemented a little more differently and you can just imagine that they are built on top of the mapmap and filter methods in collections.

Let’s look at an example and then explain what it is about:

```
val pairs: Tuple2[String, String] =
    for {
    product  <- products
    category <- product.category
    } yield(product.name, category.name)
```

The for expression said go through all the product in the lists and assign each of them to the product variable. Samely, go through all the category of product in the list and assign each of them to the category variable. At the end of the for expression, it yields a list of tuples that contain the name of product and the name of category.

For expressions are very useful when handling collections and one of the best tools to use along with maps, filter and all other data structures.

If you're looking for a more in-depth resource, here is some comprehensive documentation from Scala-Lang.org.

It's time to test your knowledge on Collections - take the quiz below!

```
1. Which of the following is not a collection:
    A. Set, because it does not keep order of its elements
    B. Tuples, because they contain a predefined number of elements
    C. String
    D. Streams, because they can produce an infinite amount of elements

2. When you execute map on a collection, you:
    A. Apply a function to each element, and modify the original collection with the
       results of such function
    B. Apply a function to each element, and produce a new collection with the results of
       such function
    C. Apply a function to the collection object, and return it
    D. Depends on the collection

3. What potentially undesirable effects will this operation have?

        List(...).toSet.toList

    A. Some elements might be dropped because Set does not allow duplicates
    B. The resulting list might not be in the same order as the original one
    C. All of the above
```

For Expressions

For Expressions are a very useful construct that can simplify manipulation of collections and several other data structures. They can be used in place of nested for loops, or to replace calls to map and flatMap in non-collection structures.

Let's get started with the concept of **For Expressions** and explains them with some good examples:

- Music label has artists
- Artists have albums
- Albums have volumes
- Volumes have tracks
- Tracks have a name and duration

We going to represent this model as nested collection types:

```
case class Label(artist:Seq[Artist], name:String)
case class Artist(albums:Seq[Album], name:String)
case class Album(volumes:Seq[Volume], name:String)
case class Volume(tracks:Seq[Track])
case class Track(name:String, duration:Int)
```

From the nested collection created using case classes in scala, we see that Label has artists and name, Artist has albums and name, Album has volumes and name, Volume has tracks and Track has a name and duration.

What we can do is get all track names from volume:

```
val volume = Volume(...)

val names: Seq[String] = volume.tracks.map(t=> t.name)
```

Or suppose we would like to get all track names from an album:

```
val album = Album(...)
val names : Seq[String] =
    album.volumes.flatMap(volume => volume.tracks.map(track => track.name))
```

Why this might be easy for small next collections using flatmaps and map to get the inner structure, it becomes hard to read with large next collections. This is when For Expressions can be very useful.

So with For Expression, we easily get all track names for an artist as follows:

```
val artist = Artist(...)

val names: Seq[String] =
    for{
        album  <- artist.albums
        volume <- album.volumes
        track  <- volume.tracks
    } yield track.name
```

and we can also add condition or filter to the For expressions like if you need the volumes size of the albums to be greater than 1:

val label = Label(...)

```
    val names: Seq[String] =
        for{
            artist <- label.artists
            album  <- artist.albums if album.volumes.sie > 1
            volume <- album.volumes
            track  <- volume.tracks
        } yield track
```

Keep in mind that while For expressions are very good for handling collection and for complex extractions, they can also be with any other data structure such as map, flatmap and filter.

And always remember that:

- Semantics of these depend on the type
- For Expressions always return a value
- Type of resulting data structure is driven by the first generator
- Type of the contained elements is driven by the yield

Other examples that can be use with For Expressions are:

- Options
- Try
- Either
- Future

Looking for more? This article has a short, but good example of for comprehension de-sugaring.

For a more comprehensive look, check out the official Scala-Lang documentation on this topic.

Before moving on to Featured Types, make sure that you have a solid grasp on For Expressions by completing the quiz below.

1. Which of the following types cannot be used on the right hand side of generators?
    a. Option
    b. Try
    c. Either
    d. Traversable
    e. String
    f. Int
    g. Future
2. What is the difference between for expressions and for loops?
    a. None, they are synonyms
    b. For loops can only be used on collections
    c. For loops have no yield, and return Unit
    d. All of the above
3. The type of value returned by a for expression is determined by:
    a. The first generator and the yield expression
    b. The first generator C.The last generator
    c. The yield expression

Featured Types

The Scala ecosystem has several types that may not be immediately familiar to you, but that are extremely useful to know. Here are three very important ones, since they are used throughout the core Scala API and many external libraries:

Option

```
Scala uses the Option type instead of null to represent non-existence of values. This article
explains options very well (and uses the term "hipsterrific startups", which shows profound
wisdom)
```

Try

```
There are no checked exceptions in Scala, but we can still use try-catch blocks to handle them.
However, the Try data structure offers a way to express possible errors within the type system.
Daniel Westheide explains Try in this article.
```

Either

```
In some scenarios, it's useful to create a container that wraps around values of two possibly
unrelated types. This is where Either can be helpful. This blog post explains the Either
structure very well.
```

Take a moment to complete the quiz below to make sure you've understood Featured Types.

```
1. The advantage of using Option instead of null is:
    A. It's more obvious to the reader when a variable may not contain a value
    B. No need to check for null on every single object
    C. Removing unexpected NullPointerExceptions
    D. All of the above

2. What is the advantage of using Try instead of regular try-catch blocks?
    A. We can handle more exceptions with Try
    B. The possibility of failure is explicit on the type.
    C. It allows us to remove checked exceptions from method definitions

3. Can we replace all uses of try-catch with Try?
    A. No, because Try has no "finally" functionality
    B. No, because Try does not handle all types of exceptions
    C. All of the above

4. The following code snippet compiles but always returns false:

        def is42(either: Either[Int,String]) = either == Success(42).

   What should be changed for it to properly check for the value 42?
    A. Replace "Either" for "Try"
    B. Replace "Success" for "Left"
    C. Replace "Success" for "Right"
    D. Change both type parameters of Either to Int
```

Next Steps

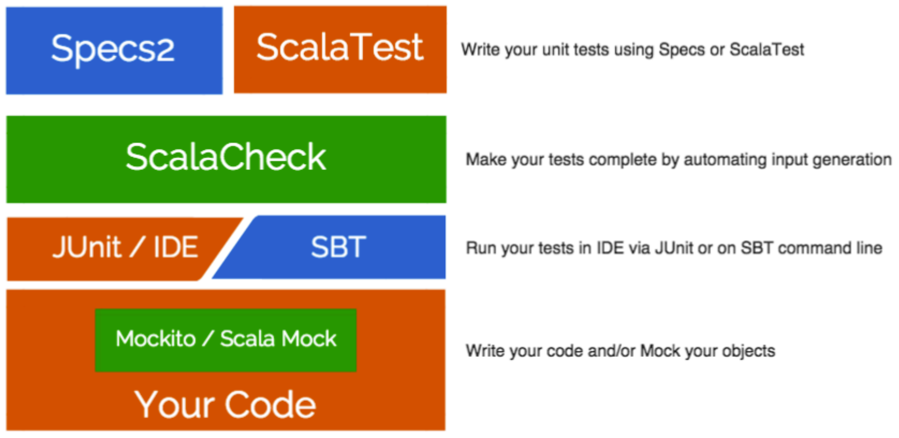
Congratulations, you've completed the basics of Scala!

Feel like you've got a good grip on everything? Are there some topics that you need some more information on? If you have any questions about any of the topics to far, don't hesitate to post in the Public Discussion forum found at the bottom of each page.

Testing & Debugging

Testing & dubugging is a big deal in Scala world. The power of the language is being heavily utilized to make behaviour driven development (BDD) aspect of the testing to be a breath of fresh air and engage developers to do test driven development (TDD) as much as possible.

This graphic shows a typical test driven flow in Scala ecosystem:


Specs2
ScalaTest
Write your unit tests using Specs or ScalaTest
ScalaCheck
Make your tests complete by automating input generation
JUnit / IDE
SBT
Run your tests in IDE via JUnit or on SBT command line
Mockito / Scala Mock
Your Code
Write your code and/or Mock your objects

Specs2 And ScalaTest

The two most important test frameworks for Scala are Specs2 and ScalaTest. While they are very much alike, there are some differences, which have been outlined on StackOverflow.

Since they provide mostly the same core features, you'll have to choose based on your own personal preference.

ScalaCheck

ScalaCheck gives you an ability to focus on your code specification by automating its testing on a large number of randomly generated cases. The test coverage significantly increases without you having to do anything manually. Though optional, it is a very useful component and is highly recommended!

You can run your tests directly in your favorite IDE (i.e. Eclipse or IntelliJ) through integration with JUnit 4 or directly with SBT, which could be used as part of your continuous integration.

If you're a fan of mocking components, you can use Mockito with both ScalaTest and Specs2. ScalaTest also integrates well with ScalaMock, JMock, or EasyMock.

Debugging

We believe the purest and more functional your code is, the less you'll need to debug it. However, you may need to at some point, in which case you'll have to choose an adequate method depending on the system you're developing. It could be a standalone SBT based background job, a web application such as Play, or part of an Akka based system.

In this section we will outline some of the standard techniques available in the Scala ecosystem today. If you don't feel ashamed to claim that **println** is the golden hammer of debugging, then you'd probably want to jump directly to the **Logging** section below and skip the first few items.

If not, let's get started:

Scala Worksheet

Use Scala Worksheet to debug parts of your code. Copy & Paste - yes, but since you're in the functional world, it should be easier to test independent components in isolation.

Eclipse: To create a worksheet in the Scala IDE, right click on a Scala project and choose: New > Scala WorkSheet. The rest will be obvious.

IntelliJ works similarly, and you'll find more details here.

REPL

Here is a nice explanation of how to run Scala in several ways, including the REPL and the aforementioned Scala worksheet. Note that we can use import statements to bring in your existing code, which make it easier to play around with existing code in your project.

IDE/Remote Debugger Eclipse: Scala IDE has a built-in Scala debugger, find details here.

IntelliJ: To debug Scala code, take a look at this. If you run into any issues, the following StackOverflow thread might be helpful.

Logging

No one likes dealing with adding logging statements in their code. However, everyone complains about the lack of important information when a problem arises and they have to troubleshoot by looking in log files.

For a fairly detailed explanation of what is available, read the following StackOverlfow thread. Regardless of which wrapper you decide to go with, if you see 'production' in your project's future at some point, you probably want to evaluate upfront which appenders are supported, for example graylog2 or something similar. Logging and instrumentation is key to understanding the behaviour of your application after-the-fact.

If you want to be really fancy with Scala and logging (i.e. you wan to use logging without compromising your hard earned one liners), you could use any one of the techniques mentioned in this thread.

Monitoring

Just like debugging, monitoring depends on type of the system you're building. Generally speaking, your stats are either being fetched by the monitoring system of your choice ('pull') or are pushed into it ('push').

The following is a short list of examples of the libraries and plugins that have some sort of Scala wrappers or integration, even if it is only valid for specific system (such as Play).

This list is not complete by any means, but it includes examples of what is available today.

1. Ostrich is an example of a Pull logging system. It's quite generic and can be used in any Scala based project. It makes use of an in-process lightweight web server.
2. This StatsD plugin integrates Play with a running StatsD server.
3. Metrics-scala Is a library that can be used in any Scala based project. It's a mix of push and pull strategies. It's JMX compatible, exposes metrics on the same server, and has integration with Graphite and Ganglia.

Support

We know that trying to learn a new language can be intimidating and challenging. Rest assured, Typesafe and YoppWorks are always here to help! In this section we'll help you understand your options when it comes to support and guidance with Scala and the Typesafe Reactive Platform.

Subscriptions

With Subscriptions, the experts at Typesafe are only a phone call or email away. In addition, subscribers get invited to Ask The Expert webinars, special perks for Scala Days, and exclusive access to the Typesafe community.

How YoppWorks Can Help

Our team of seasoned Scala experts have been working with the Lightbend Reactive Platform for years, and are here to help you in a variety of ways.

Training If you liked what you saw in this Guide and want to learn more, consider enrolling yourself in formalized Typesafe training. Our two-day training courses are taught by Typesafe certified trainers, and will give you the know-how to confidently start building applications with Scala, Akka and Play Framework. Learn more about our Typesafe training courses.

Mentorship We also offer practical training with the Lightbend Reactive Platform, which comes in the form of mentorship. Our mentorship program provides developers with an easy to manage two hours a day on hands on training, while allowing them to remain productive by working on their own projects. Learn more about our Mentorship Program.

Scala Developer Factory If your organization is looking to adopt Scala and the Typesafe Reactive Platform, we can help. Our Scala Developer Factory is the only program that can rapidly get your developers productive with Scala. The intensive two-week, hands-on, and mentor focused program will quickly give developers the practice, knowledge and habits to ensure success. Learn more about the [Scala Developer Factory].

Do you have any other questions? Feel free to contact us anytime! We'd love to chat!

Staying Up To Date

To ensure your long term success, it's important to stay up to date with what's going on in the world of Scala. We'll be doing our part by making updates and additions to this Guide as the eco-system continues to grow and evolve. Every time we make an update, you'll be notified! In the meantime, we recommend staying up to date by following the Typesafe, Scala and YoppWorks blogs.

Also, follow these Twitter accounts to stay in the know!

- @lightbend
- @Scala_lang
- @viktorklang
- @jboner
- @odersky
- @Yopp_Works