Scala Functional Programming Patterns

This book begins with the rationale behind patterns to help you understand where and why each pattern is applied. You will discover what tail recursion brings to the table and learn to create solutions without mutations. It explains the concept of memorization and infinite sequences for on-demand computation. This book also takes you through Scala’s stackable traits and dependency injection, a popular technique, for producing loosely-coupled software systems.

You will also explore how to curry favors to your code and simplify it by deconstruction via pattern matching. You will learn to do pipeline transformations using higher order functions, such as the pipes and filters pattern, further guiding you through the increasing importance of concurrent programming and the pitfalls of traditional code concurrency. Lastly, the book takes a paradigm shift to show you the different techniques that functional programming brings to your plate.

This book is an invaluable source to help you understand and perform functional programming and solve common programming problems using Scala’s programming patterns.

Who this book is written for
If you have done Java programming before and have a basic knowledge of Scala and its syntax, then this book is an ideal choice to help you to understand the context, the applicable traditional design pattern, and the Scala way. Having previous knowledge of design patterns will help, though it is not strictly necessary.

What you will learn from this book
- Get to know about functional programming and the value Scala’s FP idioms bring to the table
- Solve day-to-day programming problems using functional programming idioms
- Cut down the boiler-plate and express patterns simply and elegantly using Scala’s concise syntax
- Tame system complexity by reducing the moving parts
- Write easier to reason concurrent code using the actor paradigm and the Akka library
- Apply recursive thinking and understand how to create solutions without mutation
- Reuse existing code to compose new behavior
- Combine the object-oriented and functional programming approaches for effective programming using Scala

Grok and perform effective functional programming in Scala

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Grokking the Functional Way'
- A synopsis of the book’s content
- More information on Scala Functional Programming Patterns
About the Author

Atul S. Khot learned programming by reading C code and figuring out how it works. From there, he moved on to writing a lot of C++ code and then moved further to Java and Scala. He is an avid open source advocate who loves scripting languages and clean coding. He is ever ready to learn a new command-line trick. Atul currently works at Webonise Labs, Pune. He was also a panelist for Dr. Dobb's Jolt Awards. Last but not least, he is a trekking enthusiast and also a big foodie.
This is a book on functional programming patterns using Scala. Functional programming uses functions as basic building blocks. These are functions that don't have any side effects. This challenges our notions about how to write programs. The order of execution for such functions does not matter. We get to reason about them in a referentially transparent manner. This can be a big help in proving correctness. It feels just like a plain arithmetic problem, where \((2+2)*(3+3)\) always equals 24. You may evaluate the expression as \(2+2\) first, or as \(3+3\).

I got to know about the Unix culture early in my career. The Unix philosophy relies on pipelining small programs, each doing functionally one and only one thing. One can connect these processing nuggets together. In addition to these hundreds of ready-made building blocks, you could write your own too. These could be easily connected in the pipeline. These pipes and filters were a deeply influential concept as a whole. When I saw Scala's combinators, options, and for comprehensions, I knew I was looking at pipes and filters again. The nuggets in this case were Scala functions instead of Unix processes. Understanding functional programming gives you a new perspective on your code and programming in general. The other aspect of pipelining is that you tend to reuse them intuitively, and you also write less. Though you iterate lines of a text file in a Unix shell pipeline, you don't write any for loops. These are done for you. You just specify which lines pass your criteria or how to transform these lines or both.
Scala allows you to do just that—albeit in a somewhat different form. You don't need to write a for loop, and you keep away from loop counters. Instead, the language invites you to write for comprehensions. Immutability is an actively advocated rule of thumb. Another is avoiding side effects. Scala advocates both. As you probably know, immutability paves the way for more robust concurrency. Why are these so important? Simple, we need to reason about code. Any strategy that makes this activity controlled and simpler is a godsend! Does going down the immutable route mean we end up doing too much copying? How would this Copy On Write measure up against large data structures? Scala has an answer for this in the form of structural sharing.

One-liners are very popular as they get a lot done in a line of code. Scala features allow you to compose such one-liners. Instead of reiterating the same collection, you can do it in one elegant expression. For example, creating an immutable class with constructor and equality comparison defined that is bestowed with destructuring powers is just a one-liner. We just define a case class. There are many situations where Scala one-liners save a lot of programmer time and result in far less code. Combinators such as map, flatMap, filter, and foreach are composed together to express a lot of logic in a one-liner. How does it all affect a program design and design patterns? Here are a few illustrative cases. The singleton design pattern is used to ensure that only one instance of a class could ever exist. Null Objects are specialized singletons that are used to avoid nulls and null checks. Scala's Options give us a similar functionality. Scala's object keyword gives us ready-made singletons. Singletons are specialized factories. A factory creates objects. Scala's syntactic sugar give us a very succinct way to use the apply factory method.

The command design pattern encapsulates an object as a command. It invokes a method on the object. Scala has parameters by name. These are not evaluated at the call site but instead are evaluated at each use within the function. This feature effectively replaces the command pattern with first-class language support. The strategy pattern encapsulates algorithms and allows us to select one at runtime. In Java, we could express the strategy as an interface and the varying algorithms as concrete implementations. Scala's functions are first-class objects. You can pass functions around as method arguments and return values. Functions can be very effective substitutes for the strategy pattern. The ability to define anonymous functions is really helpful here. The Decorator pattern is needed at times. It can be used to decorate (that is, extend) the functionality of an object without modifying it. Some design plumbing needs to be done though. Scala's stackable traits can express the same design very elegantly. One use of the proxy design pattern is for implementing lazy evaluation. When some computation is expensive, we do it only when needed.
As we are very familiar with eager evaluations, we create a list in memory and think it is fully realized. However, just like eager lists, there are lazy lists too. If we think of a typical OR (||) conditional statement, if the left operand is true, the right is not evaluated. This is a very powerful concept. Scala's streams provide lazy lists.

Chain of responsibility is another handy pattern that is used to decouple the sender of a request from its receiver and allows more than one object (a chain of objects) to handle a request. If any object in the chain is not able to handle the request, it passes the request to its next object in the chain. Scala's partial functions fit this bill nicely. We can chain partial functions with Scala's orElse operator to realize the pattern.

When we write code, we need to handle errors. Scala's Try/Success/Failure again allows us to write pipelines, and if any piece of the pipeline is an error, rest of the pipeline processing is skipped.

We will look at all these concepts in greater detail. We will set up a problem, look at the traditional Java solution, and then see how Scala changes the game.

Welcome to the Scala wonderland!

What this book covers

Chapter 1, Grokking the Functional Way, gives you an eagle's eye view of functional programming and its advantages: succinct and readable code. Also, this chapter compares the command pattern in Java and Scala.

Chapter 2, Singletons, Factories, and Builders, covers singletons and Null Objects as specialized singletons. Scala Options are null objects. This also covers Scala's support for factory method and builders.

Chapter 3, Recursion and Chasing Your Own Tail, discusses the concept of recursion and Scala's support for it. It also looks at how recursion advocates immutability and the concept of structural sharing.

Chapter 4, Lazy Sequences – Being Lazy, Being Good, talks about eager versus lazy evaluation and the proxy design pattern. It also talks about Scala's streams and infinite lists.

Chapter 5, Taming Multiple Inheritance with Traits, covers Scala traits, mix-ins, and stackable modifications. It also covers dependency injection and the Cake pattern.

Chapter 6, Currying Favors with Your Code, covers lexical scope, closures, partially applied functions, and currying. This chapter also discusses the loan pattern, template method pattern, and another way to implement decorators.
Chapter 7, Of Visitors and Chains of Responsibilities, covers the Visitor pattern and its application. The other topics that are discussed are Scala's pattern matching capabilities and the chain of responsibility pattern. We will also learn Scala implementation using orElse and the collect idiom.

Chapter 8, Traversals – Mapping/Filtering/Folding/Reducing, covers iterators and functional iteration using map, filter, fold, and reduce. This chapter introduces Monads and explains ReduceLeft and ReduceRight.

Chapter 9, Higher Order Functions, discusses the strategy pattern and Scala version using higher order functions. It covers map as a functor, flatMap as a Monad, and foldLeft as Monoids. Here, you will also learn how to iterate lazy collections.

Chapter 10, Actors and Message Passing, showcases a case study to recursively grep a directory for files that contain matching text. It also covers the producer consumer pattern and the Master Slave pattern. It explains the concept of poison pills, event-driven programming, immutability, and concurrency. It also talks about Akka and Actors and how to reimplement recursive grep using Actors.

Chapter 11, It's a Paradigm Shift, teaches you how to sort in Scala and the Schwarzian transform implemented in Scala. It discusses functional error handling with Try/Success/Failure. And talks about Java Threads versus Scala's Futures. Scala's Parser Combinators are also discussed here.
Before we start learning Scala, let's first understand what functional programming (FP) is. You may have used a spreadsheet while working. In a spreadsheet, there is a bunch of equations and we enter values in the given cells for these equations. We get the answers through these equations. When you enter the same values again, you get the same answer and there are no fallouts.

At the core of FP is composition. Looking at a software system made up of parts, we build bigger parts by composing smaller parts. If you think about it, most complex systems are composed of parts, which in turn are composed of smaller parts. Functional languages give us the means to make this composition. One of the prominent functional languages that can be used for FP is Scala.

Scala in Italian means a staircase. If you look at the language's logo, you will see it's a staircase. The language acts as a staircase through which we can ascend and become better at programming. Scala also refers to the very many techniques that we can use to control the complexity of large-scale systems.

We will begin to learn Scala by looking at abstractions. We will see why abstractions are good and how Scala helps us to be abstract. Scala code is concise and expressive. A lot of the concise expression is due to functions. Functions are an important pillar of functional programming. In this chapter, we will look at pure and impure functions. Reducing the number of moving parts to be used is an effective technique to control programming complexity. Immutability is another pillar of FP that helps us here. To see all this in effect, we will take a problem example and implement the solution in Java. Then, we will solve the problem using Scala. Looking at a few Scala one-liners will help us to get started with READ/EVALUATE/PRINT LOOP (REPL). We will use Scala REPL extensively throughout the entire book.
Finally, we will look at idioms and patterns. Traditional patterns in Scala look very different from their Java counterparts. We will briefly look at the command and strategy of Scala and see how functions are used to pass algorithms around.

Welcome to the Scala fun ride!

**Abstractions**

What do we mean by abstractions? Why are they important? To understand this, we will compare two approaches. First, the "go to the wall and pull at one of the wooden panels fitted into the rectangular hole" approach versus the "open the door, please" approach.

A door is an abstraction. We really don't care whether it is made of wood or some other material, or what type of a lock it has, or any other details. The details are important, but not always. If we worry about the details always, we would quickly get bogged down, and all the communication would effectively come to a halt.

A table is an abstraction, so is a chair, and so is a house. I hope you get the drift. We want to be selectively ignorant of the details at times, and selective ignorance is abstraction.

Now, you may ask why does it matter to us programmers? The reason is that it gets things done in a compact manner.

For example, let's look at the following Scala code, which is used to count the frequency of characters in a string:

```scala
scala> "hello world".toList.filter(_.isLetter).groupBy(x => x).map { y =>
  |   y._1 -> y._2.size
  | }
res1: scala.collection.immutable.Map[Char,Int] = Map(e -> 1, l -> 3, h -> 1, r -> 1, w -> 1, o -> 2, d -> 1)
```

Isn't it compact?

On the Urban Dictionary website, http://www.urbandictionary.com/define.php?term=cutie, the term "cutie" is defined as compact beauty—the kind you just want to put in your pocket and keep beside you forever.

I was bowled over when I first saw it all. It is concise, short, packs a punch, and is elegant. The **Type Less Do More** philosophy. Gets a lot done with less...
To run the code snippet, fire up the Scala REPL. This looks a lot like a command console, a prompt waiting for you to key in Scala code. In a command terminal, typing just Scala fires up the REPL:

```scala
~> scala
```

It will give the following output:

```
Welcome to Scala version 2.11.7 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0_25).
```

You can type in expressions to have them evaluated and type `help` for more information. Let's try the following simple one line commands:

```scala
scala> 1 + 1
res2: Int = 2
scala> "hello".length
res4: Int = 5
```

For me, Scala brought the thrill back to programming... You can do a great deal by writing a few lines of code—less is more...

Scala gives you many tools, so the code is abstract and reusable...

**Concise expression**

You always want to be able to concisely express yourself. Let's try the following command to create a string:

```scala
scala> val list = List("One", "two", "three", "Four", "five")
list: List[String] = List(One, two, three, Four, five)
```

We have created a list of strings. Note that we neither had to specify the type of the list for this, nor any new keyword. We just expressed that we wanted a list assigned to a read-only variable.

Code reviews do a lot of good to a code base. I keep looking for places where I can replace a variable assignment with variable initialization. Refer to the following URL for an example: http://www.refactoring.com/catalog/replaceAssignmentWithInitialization.html.
Scala helps us with the `val` keyword. With the use of this keyword, we can establish the following:

- The initial value of variable must be specified (it is impossible for the variable to remain uninitialized).
- The value of variable cannot ever be changed again (there is one less moving part).

Why is this so important? A system with less moving parts is easier to understand and explain. You will be knowing that Google is well known for its less-moving-parts software. Let's try the following command to check for the uppercase in these characters:

```scala
def hasUpperCaseChar(s: String) = s.exists(_.isUpper)
```

What does `s.exists(_.isUpper)` do? In this code, I have a string and I am checking whether it has one or more uppercase characters.

Note that I need to look at each character of the string to arrive at an answer as output. However, I did not have to split the string into its constituent characters and then work on each character.

I am just expressing the algorithm. The algorithm involves iterating all characters. However, I did not write a loop. Instead, I expressed what I meant, concisely. Scala's strings are collections of characters. We can try the following command to make use of a `filter` method:

```scala
list filter (hasUpperCaseChar)
```

Just like a string, `List` is again a collection, in this case, of strings. I used a list method, `filter`, to weed out elements that did not satisfy the predicate.

If we were to write this imperatively, we would need a nested loop (a loop within another loop). The first loop would take each string, and the inner loop would work on each character of the string. We would need a list to collect the matching elements.

Instead of lists, we just declared what we wanted to happen. However, at some point of time in the code the looping needs to happen! It does happen indeed, but behind the scenes and in the `filter` method.

The `filter` method is a higher order function that receives the `hasUpperCaseChar` function.
Let’s say, in addition to this method, we want only those string elements that have a length greater than 3:

```scala
scala> list filter (x => hasLowerCaseChar(x) && x.size > 3)
res1: List[String] = List(Four)
```

We are again executing the algorithm; however, with a different match criteria. We are passing a function in the form of a function literal. Each element in the list is bound to \( x \), and we run a check on \( x \).

The preceding form of expression allows us to concisely express our intent. A large part of this flexibility comes from the idea of sending small computations around that are expressible without much ado. Welcome to functions!

**Functions**

Functional programming includes a lot about functions. There are different kinds of functions in programming, for example, pure functions. A pure function depends only on its input to compute the output. Let’s try the following example to make use of functions:

```scala
scala> val addThem = (x: Int, y: Int) => x + y + 1
addThem: (Int, Int) => Int = <function2>
scala> addThem(3,4)
res2: Int = 8
```

As long as the function lives, it will always give the result 8 given the input (3,4). Take a look at the following example of a pure function:

![Pure function diagram](image-url)

Figure 1.1: Pure functions
The functions worked on the input and produced the output, without changing any state. What does the phrase "did not change any state" mean? Here is an example of a not-so-pure function:

```scala
scala> var p = 1
p: Int = 1
scala> val addP = (x: Int, y: Int) => {
    | p += 1
    | x + y + p
    | }
addP: (Int, Int) => Int = <function2>

scala> addP(3, 4)
res4: Int = 9
scala> addP(3, 4)
res5: Int = 10
```

This `addP` function changes the world — this means that it affects its surroundings. In this case, the variable `p`. Here is the diagrammatic representation for the preceding code:

![Diagram of an impure function](image)

Figure 1.2: An impure function

Comparing `addThem` and `addP`, which of the two is clearer to reason about? Remember that while debugging, we look for the trouble spot, and we wish to find it quickly. Once found, we can fix the problem quickly and keep things moving.

For the pure function, we can take a paper and pen, and since we know that it is **side effects free**, we can write the following:

```scala
addThem(3, 4) = 8
    addThem(1, 1) = 3
...```
For small numbers, we can do the function computation in our heads. We can even replace the function call with the value:

```
scala> addThem(1,1) + addThem(3,4)
res10: Int = 11
scala> 3 + 8
res11: Int = 11
```

Both the preceding expressions are equivalent. When we replace the function, we deal with **referentially transparent** expressions. If the function is a long running one, we could call it just once and cache the results. The cached results would be used for the second and subsequent calls.

The `addP` function, on the other hand, is referentially opaque.

**Immutable**

Each one of us has a name. Let’s keep this simple—a first and last name. My first name is Atul and my last name is Khot. If someone suddenly called me by the name Prakash, things won’t work!

Keeping aside cases such as writers taking a pen name (that is, Plum for PG Wodehouse), commonly each one of us has a standard, official name. We simply don’t want parts of it changed to willy nilly. Let’s try the following example:

```
scala> case class FullName(firstName: String, lastName: String)
defined class FullName
scala> val name = FullName("Bertie", "Wooster")
name: FullName = FullName(Bertie,Wooster)
scala> name.firstName = "Mrs. Bertie"
<console>:13: error: reassignment to val
  name.firstName = "Albert"
```

Scala stopped us changing the code of Woosters!! It just saved Bertie from getting a wife!

In case you need a break and some light relief, Google *The Code of the Woosters!*
Once a case class instance is created, it is sealed. You can read it, but you cannot change it:

```scala
nen.name.firstName
res12: String = Bertie

nnen.lastName
res13: String = Wooster
```

You can even look at the signified version of the instance that the compiler writes for you:

```scala
name
res14: FullName = FullName(Bertie, Wooster)
```

And you can destructure it using pattern matching. Immutability just reduces the moving parts and helps us to restore sanity. This is a boon when threads enter the picture.

## Referential transparency

To understand referential transparency, let's first consider the following description:

Let me tell you a bit about India's capital, New Delhi. The Indian capital houses the Indian Parliament. The Indian capital is also home to Gali Paranthe Wali, where you get to eat the famous parathas.

We can also say the following instead:

Let me tell you a bit about India's capital, New Delhi. New Delhi houses the Indian Parliament. New Delhi is also home to Gali Paranthe Wali, where you get to eat the famous parathas.

Here, we substituted New Delhi with the Indian capital, but the meaning did not change. This is how we would generally express ourselves.

The description is referentially transparent with the following commands:

```scala
def f1(x: Int, y: Int) = x * y

f1: (x: Int, y: Int)Int
```

```scala
def f(x: Int, y: Int, p: Int, q: Int) = x * y + p * q

f: (x: Int, y: Int, p: Int, q: Int)Int
```

```scala
f(2, 3, 4, 5)
res0: Int = 26
```
If we rewrite the \( f \) method as follows, the meaning won't change:

```scala
scala> def f(x: Int, y: Int, p: Int, q: Int)= f1(x, y) + f1(p, q)
f: (x: Int, y: Int, p: Int, q: Int)Int
```

The \( f1 \) method just depends upon its arguments, that is, it is pure.

Which method is not referentially transparent? Before we look at an example, let's look at Scala's `ListBuffer` function:

```scala
scala> import scala.collection.mutable.ListBuffer
import scala.collection.mutable.ListBuffer
```

The `ListBuffer` is a mutable collection. You can append a value to the buffer and modify it in place:

```scala
scala> val v = ListBuffer.empty[String]
v: scala.collection.mutable.ListBuffer[String] = ListBuffer()
scala> v += "hello"
res10: v.type = ListBuffer(hello)
scala> v
res11: scala.collection.mutable.ListBuffer[String] = ListBuffer(hello)
scala> v += "world"
res12: v.type = ListBuffer(hello, world)
scala> v
res13: scala.collection.mutable.ListBuffer[String] = ListBuffer(hello, world)
```

Armed with this knowledge, let's now look at the following command:

```scala
scala> val lb = ListBuffer(1, 2)
lb: scala.collection.mutable.ListBuffer[Int] = ListBuffer(1, 2)
scala> val x = lb += 9
x: lb.type = ListBuffer(1, 2, 9)
scala> println(x.mkString("-"))
```

```
Grokking the Functional Way

1-2-9
scala> println(x.mkString("-"))
1-2-9

However, by substituting \(x\) with the expression \((lb += 9)\), we get the following:

scala> println((lb += 9).mkString("-")) // 1
1-2-9-9
scala> println((lb += 9).mkString("-")) // 2
1-2-9-9-9

This substitution gave us different results. The += method of ListBuffer is not a pure function as there is a side effect that occurred. The value of the \(lb\) variable at 1 and 2 is not the same.

The problem – grouping continuous integers
A while back, I wanted a simple and elegant way to solve the following problem:

**Given**: A sorted list of numbers

**When**: We group these numbers

**Then**: Each number in a group is higher by one than its predecessor. So, let's be good and write a few tests first:

```java
@Test
public void testFourGroup() {
    List<Integer> list = Lists.newArrayList(1, 2, 3, 4, 5, 9, 11, 20, 21, 22);
    List<List<Integer>> groups = groupThem.groupThem(list);
    assertThat(groups.size(), equalTo(4));
    assertThat(groups.get(0), contains(1, 2, 3, 4, 5));
    assertThat(groups.get(1), contains(9));
    assertThat(groups.get(2), contains(11));
    assertThat(groups.get(3), contains(20, 21, 22));
}

@Test
public void testNoGroup() {
    List<Integer> emptyList = Lists.newArrayList();
    List<List<Integer>> groups = groupThem.groupThem(emptyList,
```
new MyPredicate());
assertThat(groups, emptyIterable());
}

@Test
public void testOnlyOneGroup() {
List<Integer> list = Lists.newArrayList(1);
List<List<Integer>> groups = groupThem.groupThem(list,
    new MyPredicate());
assertThat(groups.size(), equalTo(1));
assertThat(groups.get(0), contains(1));
}

We will make use of the excellent Hamcrest matchers (and the excellent Guava library) to help us express succinctly what we want our code to do.

Java code
You know the drill! Let's roll up our sleeves and dish out some code. The following looks pretty good:

    public List<List<Integer>> groupThem(final List<Integer> list) {
        final List<Integer> inputList = Collections.unmodifiableList(list);
        final List<List<Integer>> result = Lists.newArrayList();
        int i = 0;
        while( i < inputList.size()){
            i = pickUpNextGroup(i, inputList, result); // i must progress
        } return result;
    }

    private int pickUpNextGroup(final int start, final List<Integer> inputList,
        final List<List<Integer>> result) {
        Validate.isTrue(!inputList.isEmpty(),
            "Input list should have at least one element");
        Validate.isTrue(start <= inputList.size(), "Invalid start index");

        final List<Integer> group = Lists.newArrayList();
        int currElem = inputList.get(start);
group.add(currElem); // We will have at least one element
   in the group

int next = start + 1; // next index may be out of range

while (next < inputList.size()) { 
   final int nextElem = inputList.get(next); // next is in range
   if (nextElem - currElem == 1) { // grouping condition
      group.add(nextElem);
      currElem = nextElem; // setup for next iteration
   } else {
      break; // this group is done
   }
   ++next;
}
result.add(group); // add the group to result list
Validate.isTrue(next > start); // make sure we keep moving
return next; // next index to start iterating from
   // could be past the last valid index
}

This code has a lot of subtlety. We use the Apache commons 3 validation API for asserting the invariants. Download this book's source code to check out the unit test.

We are using the excellent Guava library to work with lists. Note the ease of this library, with which we can create and populate the list in one line. Refer to https://code.google.com/p/guava-libraries/wiki/CollectionUtilitiesExplained for more details.

We are careful to first make the input list an unmodifiable list — to protect ourselves from stepping on our toes... We also need to carefully arrange the iteration to make sure that it eventually terminates and we do not accidentally access a non-existent list index (try to say it without gulping).

**Going scalaish**

Remember the sermon on being abstract? Our Java code is dealing with a lot of higher level and lower level details—all at the same time... We really don't want to do that; we wish to selectively ignore the details... We really don't want to deal with all the corner cases of iteration—the Java code is worried stiff about looking at two consecutive elements and the corner cases.
What if this was somehow handled for us so we can focus on the business at hand? Let’s take a look at the following example:

```scala
def groupNumbers(list: List[Int]) = {
  def groupThem(lst: List[Int], acc: List[Int]): List[List[Int]] = lst match {
    case Nil => acc.reverse :: Nil
    case x :: xs =>
      acc match {
        case Nil => groupThem(xs, x :: acc)
        case y :: ys if (x - y == 1) =>
          acc.reverse :: groupThem(xs, x :: List())
        case _ =>
          acc.reverse :: groupThem(xs, x :: List())
      }
  }
  groupThem(list, List())
}
groupNumbers(x)
```

Thinking recursively...

This version looks a lot better—doesn’t it? We use features such as nested functions and recursion and get stuff such as immutability for free... A list in Scala is immutable. In Java, we need to work a bit more to make things immutable. In Scala, it is the other way around. You need to go a little further to bring in mutability... We also use something called persistent data structures that have nothing to do with databases here. We will look at it in detail in a while...

However, a bit better version follows—meaning a tail-recursive version. Take a look at the following example of it:

```scala
def groupNumbers(list: List[Int])(f: (Int, Int) => Boolean) : List[List[Int]] = {
  @tailrec
  def groupThem(lst: List[Int], result: List[List[Int]], acc: List[Int]): List[List[Int]] = lst match {
    case Nil => acc.reverse :: result
    case x :: xs =>
      acc match {
        case Nil => groupThem(xs, result, x :: acc)
        case y :: ys if (x - y == 1) =>
          acc.reverse :: groupThem(xs, result, x :: List())
        case _ =>
          acc.reverse :: groupThem(xs, result, x :: List())
      }
  }
  groupThem(list, List(), List())
groupNumbers(x)
```
case y :: ys if (x - y == 1) =>
    groupThem(xs, result, x :: acc)

    case _ =>
        groupThem(xs, acc.reverse :: result, x :: List())
    }
}
val r = groupThem(list, List(), List())
r.reverse

The @tailrec function does some good to our code. This is an annotation that we put on a method. We use it to make sure that the method will be compiled with tail call optimization (TCO). TCO converts the recursive form into a loop.

If the Scala compiler cannot apply TCO, it flags an error. Recursion and TCO are covered in detail in Chapter 3, Recursion and Chasing Your Own Tail.

Now, having set the stage—let's look at the reusability feature...

**Reusability – the commonality/variability analysis**

Instead of a fixed grouping criteria, could we make the Java version more dynamic? Instead of expressing the condition directly in the code, we could wrap up the condition as a predicate:

```java
if (nextElem - currElem == 1) { // grouping condition
```

A predicate is a function that returns a Boolean value. It should take two numbers and tell whether the predicate is true or false?

Now there could be multiple predicates. For example, we might want to find elements that differ by 2 or 3 numbers. We can define an interface as follows:

```java
public interface Predicate {
    boolean apply(int nextElem, int currElem);
}

public class MyPredicate implements Predicate {
    public boolean apply(int nextElem, int currElem) {
        return nextElem - currElem == 1;
    }
}
```
Why would we do this? We would do this for two reasons:

- We would want to reuse the algorithm correctly and pick up two consecutive elements. This algorithm would handle all corner cases as in the previous cases.
- The actual grouping criteria in our case is \( \text{nextElem} - \text{currElem} == 1 \), but we can reuse the previous code to regroup the numbers using another criteria.

Here is how grouping would look; I am just showing the changed code:

```java
public List<List<Integer>> groupThem(List<Integer> list, Predicate myPredicate) {
    ...
    while (int i = 0; i < inputList.size();) {
        i = pickUpNextGroup(i, inputList, myPredicate, result); // i must
        // progress
        ...
    }
}

private int pickUpNextGroup(int start, List<Integer> inputList, Predicate myPredicate, List<List<Integer>> result) {
    ...
    final int nextElem = inputList.get(next); // next is in range
    // if (nextElem - currElem == 1) { // grouping condition
    if (myPredicate.apply(nextElem, currElem)) { // grouping condition
        group.add(nextElem);
    }
    ...
}
```

It is some work to define and use the predicate. Scala makes expressing this variability pretty easy, as we can use functions. Just pass on your criteria as a function and you are good to go:

```scala
def groupNumbers(list: List[Int])(f: (Int, Int) => Boolean) : // 1  = {
    def groupThem(lst: List[Int], acc: List[Int]): List[List[Int]] = lst match {
        case Nil => acc.reverse :: Nil
        case x :: xs =>
            acc match {
                case Nil => groupThem(xs, x :: acc)
In 1, we use the function - (f: (Int, Int) => Boolean).

The f: (Int, Int) => Boolean syntax means that f is a function parameter. The function takes two Int params and returns Boolean. Here is a quick REPL session to understand this better:

```
scala> val x : (Int, Int) => Boolean = (x: Int, y: Int) => x > y
x: (Int, Int) => Boolean = <function2>
scala> x(3, 4)
res0: Boolean = false
scala> x(3, 2)
res1: Boolean = true
scala> :t x
(Int, Int) => Boolean
```

We are using the :t feature of the REPL to look at the type of x.

This is a function—a bit of code we can pass around—that corresponds to the previous Java predicate snippet. It is a function that takes two integers and returns a Boolean.

(A function is in fact a class that mixes in a trait, function2 in our case).

At 2, we use it. Now comes the fun part.
At 3, we hold on to everything else except the function. And at 4 and 5, we just pass different functions. It would be more correct to say that we use function literals. Lo and behold — we get the answers right.

This code also shows some currying and partially applied functions in action… We cover both of these features in Chapter 6, Currying Favors with Your code.

The one-liner shockers

Have you heard this phrase anytime: it is just a one-liner! You see all the work done in a single line of code, it seems almost magical. The Unix shell thrives on these one-liners. For example:

```
seq 1 100 | paste -s -d '*' | bc
```

Phew! This single command line generates a sequence of numbers, 1 to 100, generates a multiplication expression from these, and feeds the same to bc calculator that does the actual multiplication.

Scala has a legion of these. Here is one applied to our problem by using scalaz library.

To try the following snippet, you need to install the scalaz library. It is not part of the Scala standard library. Here are a few simple instructions to install it.

Create a directory of your choice and switch to it. Next, download and install the Simple Build Tool (sbt) from http://www.scala-sbt.org/download.html.

```
/Users/Atulkhot/TryScalaz> sbt
[info] Set current project to tryscalaz (in build file:/Users/Atulkhot/TryScalaz/)
> set scalaVersion := "2.11.7"
... // output elided
> set libraryDependencies += "org.scalaz" %% "scalaz-core" % "7.1.0"
... // output elided
> console
...
Now the following snippet should work after installing the scalaz library:

```scala
import scalaz.syntax.std.list._
import scalaz.syntax.std.list._
scala> List(1, 2, 3, 4, 6, 8, 9) groupWhen ((x,y) => y - x == 1)
res3: List[scalaz.NonEmptyList[Int]] = List(NonEmptyList(1, 2, 3, 4), NonEmptyList(6), NonEmptyList(8, 9))
```

Using functions the solution is just a one-liner:

```scala
scala> List(1, 3, 4, 6, 8, 9) groupWhen ((x,y) => y - x == 2)
res4: List[scalaz.NonEmptyList[Int]] = List(NonEmptyList(1, 3), NonEmptyList(4, 6, 8), NonEmptyList(9))
```

So we are just not writing any of the supporting code—just stating our criteria for grouping.

You can read more about Scalaz at http://eed3si9n.com/learning-scalaz/.

## Scala idioms

When do we know a language? In English, when we say a penny for your thoughts, we are using an idiom. We can express ourselves more succinctly and natively using these. Beating around the bush is another one. If we pick up enough of these and hurl them around at times, this will makes us fluent.

It is almost the same with programming languages. You see a construct time and again and make use of notable features of a specific programming language. Here is a sample of idioms from a few prominent languages.

For example, here is an idiomatic Scala way to sum up two lists of numbers:

```scala
scala> val d1 = List(1, 2, 3, 4, 5)
d1: List[Int] = List(1, 2, 3, 4, 5)

scala> val d2 = List(11, 22, 33, 44, 55)
d2: List[Int] = List(11, 22, 33, 44, 55)

scala> (d1, d2).zipped map (_ + _)
res0: List[Int] = List(12, 24, 36, 48, 60)
```
We could do this in a roundabout way; however, note that your Scala colleagues would quickly comprehend what is happening. Try the following command:

```scala
scala> (1 to 100).map(_ * 2).filter(x => x % 3 == 0 && x % 4 == 0 && x % 5 == 0)
res2: scala.collection.immutable.IndexedSeq[Int] = Vector(60, 120, 180)
```

For numbers from 1 to 100, we multiply each number by 2. We select those numbers that are divisible by 3, 4, and 5.

Here we are chaining method calls together. Each method returns a new value. The intermediate values are threaded from call to call.

We could also write the preceding command as follows

```scala
scala> val l1 = 1 to 100
... // output elided
scala> val l2 = l1.map(_ * 2)
... // output elided
scala> val l3 = l2.filter(x => x % 3 == 0 && x % 4 == 0 && x % 5 == 0)
l3: scala.collection.immutable.IndexedSeq[Int] = Vector(60, 120, 180)
```

However, the fluent API style is more idiomatic.

People relate easily to idiomatic code. When we learn and use various Scala idioms, we will write code in the Scala way.

### Patterns and those aha! moments

Patterns have more to do with a software design such as how to compose things together, the kind of design template to be used, and so on. They are a lot more about interactions between classes and objects. Patterns are design recipes, and they illustrate a solution to a recurring design problem. However, idioms are largely specific to a language and patterns are more general and at a higher level. Patterns are also mostly language independent.

You can (and usually do) implement the same design patterns in the language you are working with.
The command design pattern

The command design pattern encapsulates an object. It allows you to invoke a method on the object at a later point. For example, we are used to the following code line:

Given a Java object:

```
    a.someMethod(arg1, arg2);
    a.method1(arg1, arg2);
    a.method2(arg3);
```

We expect the call `a.method1` to complete before the `a.method2` call starts. On the other hand, consider a real life situation.

Let's say you go to a restaurant, sit at a table, and order food. The waiter scribbles down the order on a piece of paper. This piece of paper then goes to the kitchen, someone cooks the food, and the food is served. It makes sense to prepare food for someone who ordered earlier, so your order is queued.

In the preceding paragraph, the piece of paper holds the details of your order. This is the command object. The preceding description also talks about a queue where the someone who cooks is the invoker—he puts things in motion as per your command. Add the undo() functionality and you have the essence of the command design pattern. Database transactions need to undo the commands on failure—the rollback semantics, for example.

Here is a simple first cut as example:

```scala
    def command(i : Int) = println(s"---$i---")
    def invokeIt(f : Int => Unit) = f(1)
    invokeIt(command)
```

The `def` method gets converted to a function. This is called an ETA expansion. We will soon be looking at the details.

This is a bit unpalatable though. I can possibly pass any function whatsoever and possibly wreak havoc. So, to constrain the things we can possibly pass on to the invoker, we take a bit of help from the compiler by typing the following commands:

```scala
    scala> case class Command(f: Int => Unit)
    defined class Command

    scala> def invokeIt(i: Int, c: Command) = c.f(i)
```

---

[20]
invokeIt: (i: Int, c: Command)Unit

scala> def cmd1 = Command(x => println(s"**${x}**"))
cmd1: Command

scala> def cmd2 = Command(x => println(s"++++${x}++++"))
cmd2: Command

scala> invokeIt(3, cmd1)
**3**

scala> invokeIt(5, cmd2)
++++5++++

It is so terse.

The strategy design pattern

Strategy helps us to define a set of algorithms, encapsulates each step, and selects one as appropriate.

Oh boy! Here it is. We surely have used the java.util.Comparator strategy interface at times that allows us to vary the compare algorithm as we see fit so we can sort arrays at will. Let's try the following example:

```java
Integer[] arr = new Integer[] { 1, 3, 2, 4 }; Comparator<Integer> comparator = new Comparator<Integer>() {
    @Override
    public int compare(Integer x, Integer y) {
        return Integer.compare(y, x); // the strategy algorithm – for reverse sorting
    }
}; Arrays.sort(arr, comparator);
System.out.println(Arrays.toString(arr));
```

Scala makes it a breeze by using these strategy... Type the following command to sort an array:

```scala
scala> List(3, 7, 5, 2).sortBy(_ < _)
res0: List[Int] = List(2, 3, 5, 7)
```
Passing algorithms around

We need this ability to plug into an algorithm as needed. When we start applying a strategy, we really try to apply the **Open/Closed principle (OCP)**. We don't touch the sort algorithm internals, that is, the sort implementation is closed for modification. However, by passing in a comparator, we can use the algorithm to sort objects of various classes that are open for extension.

This open for extension feature is realized very easily in Scala, as it allows us to pass functions around.

Here's another code snippet as an example of passing functions:

```scala
def addThem(a: Int, b: Int) = a + b // algorithm 1
def subtractThem(a: Int, b: Int) = a - b // algorithm 2
def multiplyThem(a: Int, b: Int) = a * b // algorithm 3

def execute(f: (Int, Int) => Int, x: Int, y: Int) = f(x, y)

println("Add: " + execute(addThem, 3, 4))
println("Subtract: " + execute(subtractThem, 3, 4))
println("Multiply: " + execute(multiplyThem, 3, 4))
```

Imagine writing this in Java. This code is possible because in Scala, we can pass functions around as objects. Furthermore, we can use a method where a function is expected:

```scala
val divideThem = (x: Int, y: Int) => x / y
println("Divide: " + execute(divideThem, 11, 5))
```

Scala's functions are first-class objects, meaning they can be sent to a method and returned from a method, just like a number or string. The ability to pass functions to other functions is a very powerful concept. It is one of the major pillars of FP, as we will soon see.
Summary

Scala is expressive and rich with tools that help us to eliminate boilerplates. It allows us to concisely express the intention of programming. Functions help us reuse the common facilities, freeing us from coding them every time.

Pure functions are simpler to reason about, as there are lot less moving parts. We can think of them in terms of referential transparency. Impure functions are hard to reason with. We saw how making things immutable also helps in reducing these moving parts.

Idioms are what make us use a language effectively. This is true for programming languages. Scala is a feature-rich functional programming language. We got a bird’s eye view of a few Scala features, such as recursion and functions.

Design patterns are a programmer’s vocabulary. Scala gives us a fresh perspective of patterns, and we saw how the use of functions makes using design patterns so very easy in Scala.

We implemented the solution to a problem in Java and Scala. We saw how succinct and expressive the Scala code is compared to its Java counterpart.

We got our feet wet in the Scala land. Let's look at these features in detail and see how Scala makes programming cool and fun again. We will start with singleton and factories. Get, set, and go!
Where to buy this book

You can buy Scala Functional Programming Patterns from the Packt Publishing website.
Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.