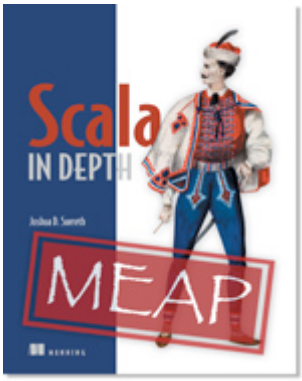
MANNING PUBLICATIONS

Examining functional concepts in Google Collections

An article from



Scala in Depth
EARLY ACCESS EDITION

Joshua D. Suereth
MEAP Release: September 2010
Softbound print: Spring 2011 | 225 pages
ISBN: 9781935182702

This article is taken from the book Scala in Depth*. The author explains how Google Collections API enhances the standard Java collections through immutable data structures and functional interaction with collections.*

The Google Collections API adds a lot of power to the standard Java collections. It brings a nice set of efficient immutable data structures and some functional ways of interacting with your collections, primarily the Function interface and the Predicate interface. These interfaces are used from the `Iterables` and `Iterators` classes. Let's take a look at the Predicate interface (listing 1) and its uses.

Listing 1 Google Collections Predicate interface

```
interface Predicate<T> {
   public boolean apply(T input);                                    #A
   public boolean equals(Object other);
}
```

#A Matches Function1.apply

The predicate interface is rather simple. Besides equality, it contains an `apply` method, which returns true or false against its argument. This is used in `Iterators`/`Iterables` `filter` method. The `filter` method takes a collection and a predicate. It returns a new collection containing only elements that pass the predicate `apply` method. Predicates are also used in the `find` method. The `find` method looks in a collection for the first element passing a Predicate and returns it.

The `filter` and `find` method signatures are shown in listing 2.

Listing 2 Iterables filter and find methods

```
class Iterables {
   public static <T> Iterable<T> filter(Iterable<T> unfiltered,
       Predicate<? super T> predicate) {...}                          #A
   public static <T> T find(Iterable<T> iterable,
       Predicate<? super T> predicate) {...}                          #B
   ...
}
```

#A Filters using predicate
#B Finds using predicate

There is also a `Predicates` class that contains static methods for combining predicates (ands/ors) as well as standard predicates for use, such as `not null`. This simple interface creates some powerful functionality through the potential combinations that can be achieved with very terse code. Also, since the predicate itself is passed into the filter function, the function can determine the best way or time to execute the filter. The data structure may be amenable to lazily evaluating the predicate, thus making the iterable return a "view" of the original collection.

It might also determine that it could best optimize the creation of the new iterable through some form of parallelism. The fact is that this has been abstracted away, so the library could improve over time with no code changes on our part.

The Predicate interface itself is rather interesting because it looks like a very simple function. This function takes some type T and returns a Boolean. In Scala, this would be represented as `T => Boolean`. Let's rewrite the `filter`/`find` methods in Scala and see what their signatures would look like (listing 3).

Listing 3 Iterables filter and find methods in Scala

```
object Iterables {
   def filter[T](unfiltered : Iterable[T],
       predicate : T => Boolean) : Iterable[T] = {...}                #A
   def find[T, U :> T](iterable : Iterable[T], predicate : U => Boolean) : T = {...}
   ...
}
```

#A No need for ?

You'll immediately notice that in Scala we aren't using any explicit `? super T` type annotations. This is because the Function interface in Scala is appropriately annotated with covariance and contravariance. Covariance (`+T` or `? extends T`) is when a type can be coerced down the inheritance hierarchy. Contravariance (`-T` or `? super T`) is when a type can be coerced up the inheritance hierarchy. Invariance is when a type cannot be coerced at all. In this case, a Predicate's argument can be coerced up the inheritance hierarchy as needed. This means, for example, that a predicate against mammals could apply to a collections of cats, assuming a cat is a subclass of mammal. In Scala, you specify co-/contra-/in-variance at class definition time.

What about combining predicates in Scala? We can accomplish a few of these rather quickly using some functional composition. Let's make a new Predicates module in Scala that takes in function predicates, and provides commonly used function predicates. The input type of these combination functions should be `T => Boolean` and the output should also be `T => Boolean`. The predefined predicates should also have a type `T => Boolean`.

Listing 4 Predicates in Scala

```
object Predicates {
  def or[T](f1 : T => Boolean, f2 : T => Boolean) =
          (t : T) => f1(t) || f2(t)                                   #A
  def and[T](f1 : T => Boolean, f2 : T => Boolean) =
          (t : T) => f1(t) && f2(t)                                   #B
  val notNull[T] : T => Boolean = _ != null
}
```

#A explicit anonymous function
#B placeholder function syntax

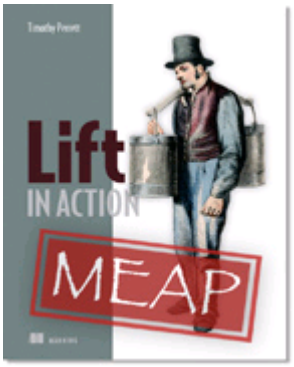
We've now started to delve into the realm of functional programming. We are defining first-class functions and combining them to perform new behaviors. You'll notice the `or` method takes two predicates, `f1` and `f2`. It then creates a new anonymous function that takes an argument `t` and *ors* the results of `f1` and `f2`.

This is the essence of functional programming. Playing with functions also makes more extensive use of generics and the type system. Scala has put forth a lot of effort to reduce the overhead for generics in daily usage.

Summary

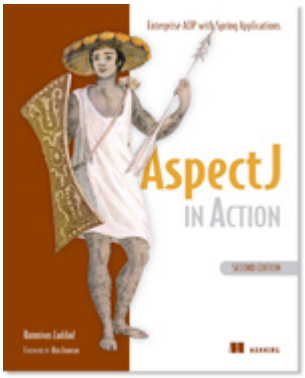
As Scala blends together various concepts, users of Scala will find themselves striking a delicate balance between functional programming techniques, object orientation, integration with existing Java applications, Expressive library APIs, and enforcing requirements through the type system.

Here are some other Manning titles you might be interested in:



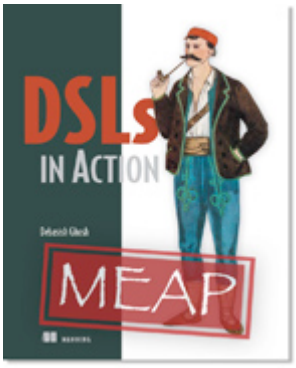
Lift in Action
EARLY ACCESS EDITION

Timothy Perrett
MEAP Release: April 2010
Softbound print: February 2011 | 450 pages
ISBN: 9781935182801



AspectJ in Action, Second Edition
IN PRINT

Ramnivas Laddad
MEAP Release: October 2009
September 2009 | 568 pages
ISBN: 1933988053



DSLs in Action
EARLY ACCESS EDITION

Debasish Ghosh
MEAP Began: October 2009
Softbound print: December 2010 (est.) | 375 pages
ISBN: 9781935182450