Scala Concepts

Tail Recursion

Functions which call themselves as their last action are called **tail-recursive**.

The Scala compiler detects tail recursion and replaces it with a jump back to the beginning of the function, after updating the function parameters with the new values ... as long as the last thing you do is calling yourself, it's automatically tail-recursive (i.e., optimized).

You can add **`@tailrec`** annotation to enforce tail recursion to a function. If the recursive function, that is annotated using `@tailrec`, is not implemented using tail-recursive algorithm then it will not compile and throws errors.

Scala's tail-recusrsion is a compilation optimization technique. Often, a recursive solution is more elegant and concise than a loop-based one. If the solution is tail recursive, there won't be any runtime overhead to be paid.

Closures

A ***closure*** is a function, whose return value depends on the value of one or more variables declared outside this function. Such variables whose scope lies outside a functions immediate lexical scope are called "free variables".

The function value (the object) that's created at runtime from this function literal is called a closure. The name arises from the act of "closing" the function literal by "capturing" the bindings of its free variables.

Any function literal with free variables, such as `(x: Int) => x + more`, is an ***open term***. Therefore, any function value created at runtime from `(x: Int) => x + more` will by definition **require that a binding for its free variable**, more, be captured. The resulting function value, which will contain a reference to the captured more variable, is called a ***closure***, therefore, because the function value is the end product of the act of closing the open term, `(x: Int) => x + more`.

Monads

Monads are structures that represent sequential computations. A Monad is not a class or a trait; it is a concept.

A Monad is an object that wraps another object in Scala.

In Monads, the output of a calculation at any step is the input to other calculations, which run as a parent to the current step.

```
val numList1 = List(1,2)
val numList2 = List(3,4)
//monad
val numlist3 = numList1.flatMap{ x => numList2.map{ y => x + y } }
```

You can formulate functions `map`, `flatMap`, and `withFilter` on a monad. Furthermore, you can characterize every monad by map, `flatMap`, and `withFilter`, plus a “unit” constructor that produces a monad from an element value. In an object-oriented language, this “unit” constructor is simply an instance constructor or a factory method. Therefore, map, `flatMap` and `withFilter` can be seen as an object-oriented version of the functional concept of monad. Because ‘for expressions’ are equivalent to applications of these three methods, they can be seen as syntax for monads.

Implicits

Implicits in Scala can mean two things:

- **Implicit Parameters**: A value that can be passed “automatically”
- **Implicit Conversion**: A conversion from one type to another that is made automatically.

Implicit Conversion

Speaking very briefly about the latter type, if one calls a method `m` on an object `o` of a class `C`, and that class does not support method `m`, then Scala will look for an implicit conversion from `C` to something that does support `m`. A simple example would be the method map on String:

```
"abc".map(_.toInt)
```

String does not support the method `map`, but `StringOps` does, and there's an implicit conversion from `String` to `StringOps` available. (see `implicit def augmentString` on `Predef`).

Implicit Parameters

The other kind of implicit is the *implicit parameter*. These are passed to method calls like any other parameter, but the compiler tries to fill them in automatically. If it can't, it will complain. (One can pass these parameters explicitly also)

- **Check out Chapter 21 (Implicit Conversions & Parameters) Programming Scala Pg 479**

Type Classes

Type classes are a powerful and flexible concept that adds ad-hoc polymorphism to Scala. The combination of **parameterized types** and **implicit parameters** is also called **type classes**.

Type class is a class (group) of types, which satisfy some contract defined in a trait with addition that such functionality (trait and implementation) can be added without any changes to the original code. One could say that the same could be achieved by extending a simple trait, but with type classes it is not necessary to predict such a need beforehand.

Variance

Variance defines Inheritance relationships of Parameterized Types. Variance is all about Sub-Typing.

A type parameter of a class or trait can be marked with a variance annotation, either *covariant* (+) or *contravariant* (). Such variance annotations indicate how sub-typing works for a generic class or trait.

For example, the generic class List is covariant in its type parameter, and thus `List[String]` is a subtype of `List[Any]`. By default, i.e., absent a + or annotation, type parameters are nonvariant.

Type Parameter:

- **`List[T]` => `T`** is called type-parameter; **`List[T]`** is called a generic

For `List[T]`, if we use `List[Int]`, `List[AnyVal]` etc. then these `List[Int]` and `List[AnyVal]` are known as "Parameterized Types". Variance defines Inheritance relationship between these Parameterized Types.

Advantage of Variance in Scala

The main advantage of Scala Variance is:

- Variance makes Scala collections more Type-Safe.
- Variance gives more flexible development.
- Scala Variance gives us a technique to develop Reliable Applications.

Types of Variance in Scala

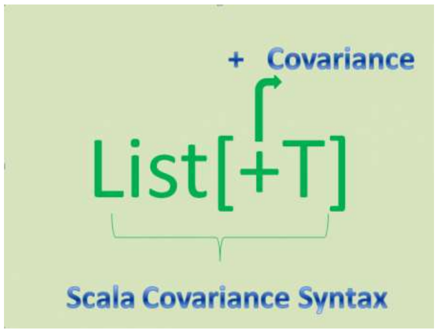
Scala supports the following three kinds of Variance.

- Covariant
- Invariant
- Contravariant

Covariant in Scala

If “S” is subtype of “T” then List[S] is is a subtype of List[T]. This kind of inheritance relationship between two Parameterized Types is known as “Covariant”

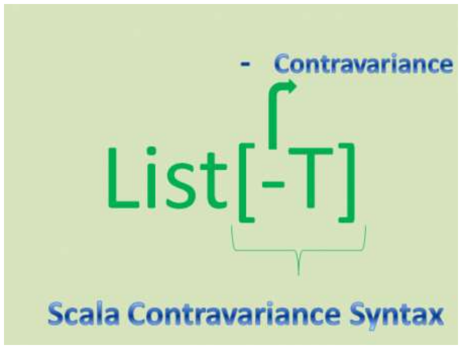
To represent Covariance relationship between two Parameterized Types, Scala uses the following syntax: **Prefixing Type Parameter with “+” symbol defines Covariance** in Scala. Here T is a Type Parameter and “+” symbol defines Scala Covariance.



Contravariant in Scala

If “S” is subtype of “T” then List[T] is is a subtype of List[S]. This kind of inheritance relationship between two Parameterized Types is known as “Covariant”

To represent Contravariance relationship between two Parameterized Types, Scala uses the following syntax: **Prefixing Type Parameter with “-” symbol defines Contravariance** in Scala. Here T is a Type Parameter and “-” symbol defines Scala Contravariance.



Invariant in Scala

If “S” is subtype of “T” then `List[S]` and `List[T]` don’t have Inheritance Relationship or Sub-Typing. That means both are unrelated.

This kind of Relationship between two Parameterized Types is known as “**Invariant** or **Non-Variant**”.

In Scala, by default Generic Types have Non-Variant relationship. If we define Parameterized Types without using “+’ or “-” symbols, then they are known as Invariants.

Variance Annotation means defining “+” or “-” before Type Parameters.

Summary:

• Covariant	[+T]	If S is subtype of T, then List[S] is also subtype of List[T]
• Contravariant	[-T]	If S is subtype of T, then List[T] is also subtype of List[S]
• Invariant	[T]	If S is subtype of T, then List[S] and List[T] are unrelated.

Extractors

Extractors can be used to define patterns that are decoupled from an object’s representation. Extractors let you define new patterns for preexisting types, where the pattern need not follow the internal representation of the type.

An *extractor* in Scala is an object that has a method called `unapply` as one of its members. The purpose of that `unapply` (which is called an *extractor*) method is to match a value and take it apart. Often, the extractor object also defines a dual method `apply` (which is called an *injector*) method for building values, but this is not required.

Whenever pattern matching encounters a pattern referring to an extractor object, it invokes the extractor’s `unapply` method on the selector expression.

`selectorString match { case EMail(user, domain) => ... }` would lead to the call: `EMail.unapply(selectorString)`

Extractors versus case classes

Case classes expose the concrete representation of data. This means that the name of the class in a constructor pattern corresponds to the concrete representation type of the selector object.

Extractors break this link between data representations and patterns. This property is called ***representation independence***.

Representation independence is an important advantage of extractors over case classes. On the other hand, case classes also have some advantages of their own over extractors.

First, they are much easier to set up and to define, and they require less code.

Second, they usually lead to more efficient pattern matches than extractors, because the Scala compiler can optimize patterns over case classes much better than patterns over extractors. This is because the mechanisms of case classes are fixed, whereas an `unapply` or `unapplySeq` method in an extractor could do almost anything.

Third, if your case classes inherit from a sealed base class, the Scala compiler will check your pattern matches for exhaustiveness and will complain if some combination of possible values is not covered by a pattern. No such exhaustiveness checks are available for extractors.

Regular Expressions

One particularly useful application area of extractors are regular expressions. Like Java, Scala provides regular expressions through a library, but extractors make it much nicer to interact with them

Scala’s regular expression class resides in package **scala.util.matching**

Cake & Magnet Patterns

Cake Pattern is **methodology of dependency injection**, which in Scala, is implemented using traits. In traits, behaviors are defined, but implemented at runtime.

Asynchronous Programming with Scala

An asynchronous computation is any task, thread, process, node somewhere on the network that:

- executes outside of your program's main flow or from the point of view of the caller, it doesn't execute on the current call-stack
- receives a callback that will get called once the result is finished processing
- it provides no guarantee about when the result is signaled, no guarantee that a result will be signaled at all

Future

A Future is an object holding a value which may become available at some point. This value is usually the result of some other computation:

1. If the computation has not yet completed, we say that the Future is not completed.
2. If the computation has completed with a value or with an exception, we say that the Future is completed.

Completion can take one of two forms:

1. When a Future is completed with a value, we say that the future was successfully completed with that value.
2. When a Future is completed with an exception thrown by the computation, we say that the Future was failed with that exception.

A **Future** has an important property that it may only be assigned once. Once a Future object is given a value or an exception, it becomes in effect immutable – it can never be overwritten.

The simplest way to create a future object is to invoke the **Future.apply** method which starts an asynchronous computation and returns a future holding the result of that computation. The result becomes available once the future completes.

Note that **Future[T]** is a type which denotes future objects, whereas **Future.apply** is a method which creates and schedules an asynchronous computation, and then returns a future object which will be completed with the result of that computation.

```
import scala.concurrent._
import ExecutionContext.Implicits.global

val firstOccurrence: Future[Int] = Future {
    val source = scala.io.Source.fromFile("myText.txt")
    source.toSeq.indexOfSlice("myKeyword")
}
```

The line import **ExecutionContext.Implicits.global** above imports the default global execution context. Execution contexts execute tasks submitted to them, and you can think of execution contexts as thread pools. They are essential for the Future.apply method because they handle how and when the asynchronous computation is executed.

Callbacks

A callback is called asynchronously once the future is completed. If the future has already been completed when registering the callback, then the callback may either be executed asynchronously, or sequentially on the same thread.

The most general form of registering a callback is by using the `onComplete` method, which takes a callback function of type `Try[T] => U`. The callback is applied to the value of type `Success[T]` if the future completes successfully, or to a value of type `Failure[T]` otherwise.

The `Try[T]` is similar to `Option[T]`. `Try[T]` is a `Success[T]` when it holds a value and otherwise `Failure[T]`, which holds an exception.

```
import scala.util.{Success, Failure}

val f: Future[List[String]] = Future {
  session.getRecentPosts
}

f onComplete {
  case Success(posts) => for (post <- posts) println(post)
  case Failure(t) => println("An error has occurred: " + t.getMessage)
}
```

The `onComplete` method is general in the sense that it allows the client to handle the result of both failed and successful future computations. In the case where only successful results need to be handled, the `foreach` callback can be used:

```
val f: Future[List[String]] = Future {
  session.getRecentPosts
}

f foreach { posts =>
  for (post <- posts) println(post)
}
```

The `onComplete` and `foreach` methods both have result type `Unit`, which means invocations of these methods cannot be chained.

When exactly a callback gets called?

Since a callback requires the value in the future to be available, it can only be called after the future is completed. However, there is no guarantee it will be called by the thread that completed the future or the thread which created the callback. Instead, the callback is executed by some thread, at some time after the future object is completed. We say that the callback is executed ***eventually***.

Furthermore, the order in which the callbacks are executed is not predefined, even between different runs of the same application. In fact, the callbacks may not be called sequentially one after the other, but may concurrently execute at the same time.

Callback Semantics

- Registering an onComplete callback on the future ensures that the corresponding closure is invoked after the future is completed, eventually.

- Registering a foreach callback has the same semantics as onComplete, with the difference that the closure is only called if the future is completed successfully.

- Registering a callback on the future which is already completed will result in the callback being executed eventually (as implied by 1).

- In the event that multiple callbacks are registered on the future, the order in which they are executed is not defined. In fact, the callbacks may be executed concurrently with one another. However, a particular ExecutionContext implementation may result in a well-defined order.

- In the event that some of the callbacks throw an exception, the other callbacks are executed regardless.

- In the event that some of the callbacks never complete (e.g. the callback contains an infinite loop), the other callbacks may not be executed at all. In these cases, a potentially blocking callback must use the blocking construct (see below).

- Once executed, the callbacks are removed from the future object, thus being eligible for GC.

Promises

So far we have only considered Future objects created by asynchronous computations started using the Future method. However, futures can also be created using promises.

While futures are defined as a type of read-only placeholder object created for a result which doesn't yet exist, a promise can be thought of as a writable, single-assignment container, which completes a future.

That is, a promise can be used to successfully complete a future with a value (by "completing" the promise) using the success method. Conversely, a promise can also be used to complete a future with an exception, by failing the promise, using the failure method.