Cafe - Scala fundamentals (part 11)

Aims

We will be covering the following topics:

- Concurrency
- Futures
- Callback

Throughout this session, you can refer to the following helpful cheatsheet which is a nice guide to support you with the Scala syntax:
[Scalacheat | Scala Documentation]

Prerequisites

You have completed Scala fundamentals part 10 You are familiar with `ScalaTest` and know how to implement a unit test. You are comfortable running tests through `sbt test`.

Concurrency

In most programming languages these days there is a way to implement *Concurrency*.

> Concurrency is defined as two or more events happening or existing at the same time. As a concept, it is the opposite of consequential, which means events that follow one after another - in sequence.
>
> You can think of this in the following example: I could be working in a kitchen and be concurrently boiling a kettle and grinding coffee beans. There is no need to perform those two actions as discrete, sequential events, as they do not depend on each other. They can be done simultaneously, or better referred as **asynchronously**.

Scala achieves this by providing two implementations, `Futures` and `Actors`. In Java, this is achieved by the `Thread` package and implementing `Runnable`.
In this section, we will be covering in-depth `Futures` and we will work through a TDD example of how to implement them referring to a real world example.

We're going to implement a Cafe application that can make a cappuccino for a Person, we will be writing Unit Tests and consolidating the Red, Green, Refactor pattern shown in the previous section.

Scala's `Future` is a nice abstraction that allows us to implement *concurrent* code in a Non-Blocking manner which utilises *callbacks*.

Further reading

Blocking versus Non-blocking code

Futures

A `Future` is another container type in `Scala` and it represents a value that may be available in the future. The `Future` is either not complete, in that some computation is still taking place, or it is completed, at which point it contains either a value or an exception.

Unlike the container types we've looked at previously, such as `Option`, `Try` and `Either`, we don't know when a `Future` will complete. When we use a `Future`, we accept that it will complete *at some point*, and at that undetermined point we'll operate on it. This allows us to have code that executes concurrently, rather than consecutively.

Further reading

scala.concurrent.Future

We can declare a `Future` in much the same way as our previous data types:

```
// Create a future
val future : Future[String]= Future {
  "My " + "First " + "Future."
}

// Print the result
future onSuccess {
  case _  => println _
}
```

In this simple example, we call `onSuccess` on our `Future` and, assuming the `Future` has completed successfully, the contents of the `Future` are printed.

We can operate on the `Future` using familiar methods like `map`, `flatMap` and `filter`, and also use a *for comprehension* to iterate on it, like we did with our `Try` examples. If we `map` on a `Future`, the expression that we pass to `map` is executed when the `Future` completes successfully.

One scenario when we might want to use a `Future` is if our application is making API calls or calling to other services, and relies upon the results of those calls to perform some operation. Let's say, for example, that we have a database that our `PetRepository` makes a call to, in order to access some data, this will then asynchronously search the database and return the data. When the data is found it resolves the `Future` and expression after the `map` function.

A `Future` can resolve to return a `Success()`. A `Future` can also have a `Failure()` state which can be handled in the following way:

```
import scala.concurrent.Future

val future : Future[String] = Future {
  s"My first future"
}

future onSuccess {
  case _ => println _
}

future onFailure {
  case e => println(_.getMessage())
}
```

As you can see, `onFailure` accepts a partial function as an argument, we have supplied this in the case of a Match statement, the argument type is a `Throwable`, which you have came across before. As you will have noticed, this is now *deprecated* in version 2.12.0 of Scala.

It is recommended to use `onComplete`:

```
future onComplete {
  case Success(v) => println(v)
  case Failure(e) => println(e)
}
```

We can also call `foreach` to traverse the Future as we would with a `List`:

```
val someFuture: Future[Missile] = …
someFuture.foreach(_.neutralize(PERMANENTLY))
someFuture.foreach(_.launch(target))
```

In the example above, we don't which `foreach` function is going to execute and in which order. It's likely that it will be different each time you execute the code. *We wouldn't want this in the real world! We don't want to launch a missile that hasn't been neutralised*

Further reading

Futures in Scala 2.12.x

You have to be careful when using Futures to ensure you have implemented your flow control correctly, however, there are also negatives to implementing all code in a synchronous manner.
Let's look at why synchronous code can be bad:

Suppose you want to prepare a cappuccino. You could simply execute the following steps, one after another, waiting until you had fully completed one before starting the next:

1. Grind the required coffee beans
2. Heat some water
3. Brew an espresso using the ground coffee and the heated water
4. Froth some milk
5. Combine the espresso and the frothed milk to a cappuccino

```
import scala.util.Try
// Some type aliases, just for getting more meaningful method signatures:
type CoffeeBeans = String
type GroundCoffee = String
case class Water(temperature: Int)
type Milk = String
type FrothedMilk = String
type Espresso = String
type Cappuccino = String
// dummy implementations of the individual steps:
def grind(beans: CoffeeBeans): GroundCoffee = s"ground coffee of $beans"
def heatWater(water: Water): Water = water.copy(temperature = 85)
def frothMilk(milk: Milk): FrothedMilk = s"frothed $milk"
def brew(coffee: GroundCoffee, heatedWater: Water): Espresso = "espresso"
def combine(espresso: Espresso, frothedMilk: FrothedMilk): Cappuccino = "cappuccino
// some exceptions for things that might go wrong in the individual steps
// (we'll need some of them later, use the others when experimenting
// with the code):
case class GrindingException(msg: String) extends Exception(msg)
case class FrothingException(msg: String) extends Exception(msg)
case class WaterBoilingException(msg: String) extends Exception(msg)
case class BrewingException(msg: String) extends Exception(msg)
// going through these steps sequentially:
def prepareCappuccino(): Try[Cappuccino] = for {
  ground <- Try(grind("arabica beans"))
  water <- Try(heatWater(Water(25)))
```

```
    espresso <- Try(brew(ground, water))
    foam <- Try(frothMilk("milk"))
  } yield combine(espresso, foam)
```

Westheide, Daniel. "Daniel Westheide." The Neophyte's Guide to Scala Part 8: Welcome to the Future - Daniel Westheide, danielwestheide.com/blog/2013/01/09/the-neophytes-guide-to-scala-part-8-welcome-to-the-future.html.

In the example above, we have implemented a number of functions that return a `String` to signify the state of the Cappuccino making process.

We have implemented a `prepareCappuccino()` function which returns a `Try[Cappuccino]`, this will resolve to either a `Success` or `Failure` as discussed before. We are then iterating over the `Try` statements and storing the value into the left associated variable, i.e. `ground`.

In this instance, it will only continue to evaluate `heatWater`, if and only if the `ground` Try resolved successfully.

Writing this function in a sequential manner has a number of benefits:

- You get a very readable step by step instruction of what to do
- You are clear in which order the statements are evaluated and you have full flow control

However, in a real world scenario, preparing a cappuccino in this manner is a very ineffective process, i.e. whilst waiting for the coffee to grind, you are blocked from performing any other task. In reality, we could be boiling water and frothing milk at the same time.

We are not effectively using all of our resources, when we can clearly perform many of these tasks concurrently.

> It's really no different when writing a piece of software. A web server only has so many threads for processing requests and creating appropriate responses. You don't want to block these valuable threads by waiting for the results of a database query or a call to another HTTP service. Instead, you want an asynchronous programming model and non-blocking IO, so that, while the processing of one request is waiting for the response from a database, the web server thread handling that request can serve the needs of some other request instead of idling along.

Westheide, Daniel. "Daniel Westheide." The Neophyte's Guide to Scala Part 8: Welcome to the Future - Daniel Westheide, danielwestheide.com/blog/2013/01/09/the-neophytes-guide-to-scala-part-8-welcome-to-the-future.html.

How do `Futures` allow us to work concurrently?

A `Future` is a write-once container. After the `Future` has been completed it effectively becomes immutable. Also, the `Future` type only provides an interface for *reading* the value to be computed. The task of *writing* the computed value is achieved via a `Promise`.

When using a `Future`, you need to provide an `ExecutionContext` to specify which `Thread` to execute the `Future` on. You can specify a custom `ExecutionContext` if need be, but *normally* the `ExecutionContext.global` (default) will suffice. This can be imported by `import ExecutionContext.Implicits.global`

Note that `Future[T]` is a type which denotes future objects, whereas `Future.apply` is a method which creates and schedules an asynchronous computation.

The signature for the `Future.apply` method is:

```
object Future {
  def apply[T](body: => T)(implicit ex: ExecutionContext): Future[T]
}

val futureThing = Future {
  ...
}
```

The above example calls the `apply` method on the companion object.

You will notice in the example above the `body` argument is being passed by-name. This means it will be evaluated when `body` is used. Not before. This is denoted by the `body : => T` syntax. Up to now, you have used call by-value which is `body : T`. This means that `body` is evaluated before it's passed into the function.

Let's see how we would brew a cappuccino in an asynchronous, concurrent way:

```
import scala.concurrent.future
import scala.concurrent.Future
import scala.concurrent.ExecutionContext.Implicits.global
import scala.concurrent.duration._
import scala.util.Random

def grind(beans: CoffeeBeans): Future[GroundCoffee] = Future {
  println("start grinding...")
  Thread.sleep(Random.nextInt(2000))
  if (beans == "baked beans") throw GrindingException("are you joking?")
  println("finished grinding...")
```

```
    s"ground coffee of $beans"
  }

  def heatWater(water: Water): Future[Water] = Future {
    println("heating the water now")
    Thread.sleep(Random.nextInt(2000))
    println("hot, it's hot!")
    water.copy(temperature = 85)
  }

  def frothMilk(milk: Milk): Future[FrothedMilk] = Future {
    println("milk frothing system engaged!")
    Thread.sleep(Random.nextInt(2000))
    println("shutting down milk frothing system")
    s"frothed $milk"
  }

  def brew(coffee: GroundCoffee, heatedWater: Water): Future[Espresso] = Future {
    println("happy brewing :)")
    Thread.sleep(Random.nextInt(2000))
    println("it's brewed!")
    "espresso"
```

In the example above, we have refactored the implementation of each function to wrap them in a Future. This will spin up a new `Thread` in an `ExecutionContext` and perform this asynchronously. Once this has completed, it will call back with the result, either a `Success` or `Failure`.

Further reading

[What is a 'Thread'?]

You will notice the `Thread.sleep` calls, this will pause the execution of the code on that `Thread` for a period of time. If we implemented the above code, you would notice the order of the `println` statements would be different every time.

Let's see how we would call the asynchronous functions above:

```
  def prepareCappuccinoSequentially : Future[Cappuccino] = {
    for {
      ground <- grind("arabica beans")
      water <- heatWater(Water(82))
      foam <- frothMilk("whole milk")
      espresso <- brew(ground, water)
    } yield {
      Cappuccino(foam, espresso)
    }
  }
```

```scala
Cafe.prepareCappuccinoSequentially map {
  c =>
    println(c)
} recover {
  case NonFatal(e) => println(s"Didn't make cappuccino $e")
}
```

In the example above, we have used a for comprehension to iterate over the result of each `Future`, storing the eventual result into variables `ground, water, foam...`. Once each `Future` has completed we then yield a `Cappuccino` back combining the two ingredients together. This returns a `Future[Cappuccino]`.

This is the first time we have discovered the `recover` keyword. This behaves like a try/catch statement, in that it will catch any subclass of `Throwable`, allowing you to handle this and return a default / alternative value.

~Question~ - What do you think will happen in the example above?

> The *for comprehension* example above will actually perform each method one after another in sequence. This is due to the Future being *instantiated* within the for comprehension. It does not know about the next Future in the chain until the first has completed, it will then evaluate the next expression and instantiate the Future. Repeating as it goes.

We can overcome the problem described above by instantiating the Futures before the for comprehension:

```scala
def prepareCappuccino(): Future[Cappuccino] = {
  val groundCoffee = grind("arabica beans")
  val heatedWater = heatWater(Water(20))
  val frothedMilk = frothMilk("milk")
  for {
    ground <- groundCoffee
    water <- heatedWater
    foam <- frothedMilk
    espresso <- brew(ground, water)
  } yield Cappuccino(foam, espresso)
}
```

This will ensure that each `Future[T]` is instantiated and begin to be executed concurrently. If we ran this example you would see it is non-deterministic. The only guarantee is that we receive the Cappuccino last.

What you may not know is that a for comprehension is just another representation for nested `flatMap` calls:

```
val nestedFuture: Future[Future[Boolean]] = heatWater(Water(25)).map {
  water => temperatureOkay(water)
}
val flatFuture: Future[Boolean] = heatWater(Water(25)).flatMap {
  water => temperatureOkay(water)
}
```

In above example, the first implementation returns a nested `Future`, this is because `temperatureOkay` returns a `Future` in this instance, this has been done by mapping the result of one Future into another type. As shown by `[Future[Future[Boolean]]]`.

Here is another example of mapping the type of one `Future` to another:

```
val temperatureOkay : Future[Boolean] = heatWater(Water(25)) map {
  water =>
    println("we're in the future!")
    (80 to 85).contains(water.temperature)
}
```

Here we have waited for the `heatWater()` `Future[T]` to succeed then we have mapped that value and transformed this to another type, in this instance `Future[Boolean]`.

In our `flatMap` example, we have achieved the same thing, however, when we use `flatMap` on the result of the first `Future`, it passes through the returned value of `temperatureOkay`, removing the additional `Future` that has been returned by the anonymous function.

You could repeat this process and `map` the result of `temperatureOkay` and transform this to yet another type, passing each result back up the chain.

However, this becomes messy very quickly and is known as *callback hell*. It should be avoided where possible, as shown above a *for comprehension* is a nice way to avoid *callback hell*.

Further reading

What is callback hell?

::**Note**: If you map over a Future and it throws an Exception in it's implementation, you will never invoke the statement that follows after the `map` keyword. Be careful of this is you have critical logic you need to perform, but the Future you receive can throw a Future.::

Phew, that is a lot to take in! Grab a coffee.

We will be moving into a paired programming exercise next, consolidating everything we have learned so far, implementing a Cafe class in a TDD manner with Futures.

Task *3 hours, paired programming*

There is an effective time management technique that many developers use when doing paired programming. This is known as the **Pomodoro Technique**.

> The technique uses a timer to break down intervals, traditionally 25 minutes in length, separated by short breaks. These intervals are named pomodoros, the plural in English of the Italian word pomodoro (tomato), named after the tomato-shaped kitchen timer.
>
> 1. Decide on the task to be done
> 2. Set the pomodoro timer (traditionally to 25 minutes)
> 3. Work on the task
> 4. End work when the timer rings, keep a count of has many periods you have completed
> 5. If you have fewer than 4 periods, take a short break (3 to 5 minutes), repeat from step 2
> 6. After four pomodoros, take a longer break (15 minutes), reset your counter, start from step 1

Adopting the Pomodoro technique and **TDD**, implement the following requirements:

Implement a `Cafe` class that has the following methods:

- `def heat(water: Water, temperature: Double = 40D): Future[Water]`
- `def grind(beans: CoffeeBeans): Future[GroundCoffee]`
- `def frothMilk(milk: Milk): Future[FrothedMilk]`
- `def brew(water: Water, coffee: GroundCoffee): Future[Coffee]`

On your first iteration, implement your solution in a sequential manner (do not use Futures). Once you have successfully brewed a coffee, refactor your solution to use Futures. You will also have to refactor your tests!

Your Cafe should do the following when:

Boiling the water

- It should `heat()` water, which will return a new instance of Water with a specified

temperature
- If no temperature is specified, then the water should always boil to 40 degrees

Grinding the coffee beans

- It should grind `CoffeeBeans`, where we have at least one concrete type `ArabicaBeans`
- It should return `GroundCoffee`, where we can ensure the beans used were Arabica beans

Frothing the milk

- It should be able to froth `WholeMilk` which is a concrete implementation of `Milk`
- It should throw an `IllegalArgumentException` when attempting to froth `SemiSkimmedMilk`
- It should return `FrothedMilk` where we can determine the type of milk that was used

Brewing the coffee

- It should return a `BrewingException` when the temperature of the water is less than 40 degrees, I should be told `The water is too cold`
- I should get a `Coffee` when I brew `Water` and `GroundCoffee` and the temperature is 40 degrees or above

Adding milk

- I should be able to add `FrothedMilk` to my coffee if I want
- The temperature of the coffee should reduce by 5 degrees when milk is added
- I should be able to determine if a Coffee has had milk added

In order to achieve this, you will need to:

- Implement a Scala `App` or `main` method in order to execute every part of your solution and brew a coffee.
- At first, you will do this by instantiating classes, and calling each method in a sequential manner. Once complete, refactor to use Futures for each steps of the coffee making process, you can then **yield** a coffee back. When testing futures, the below guide will help you on how assert a `Future`.

When you run your solution you should see the following output on the console:

`You have brewed the following coffee: Coffee at 35 degrees with Whole milk`.

If the coffee does not have milk we should see:

`You have brewed the following coffee: Coffee at 35 degrees without milk`.

Further reading

Asynchronous testing - ScalaTest

Recap

To recap, we have covered `Futures` in depth in this section. We have discussed how a `Future` is a container type that has two states `Success` or `Failure`.
We have looked at the difference between synchronous and asynchronous code and how this would be applied in a real world situation.

Resources

- Blocking versus Non-blocking code
- Welcome to the future
- Futures in Scala 2.12.x
- What is a 'Thread'?
- What is callback hell?
- Asynchronous testing - ScalaTest