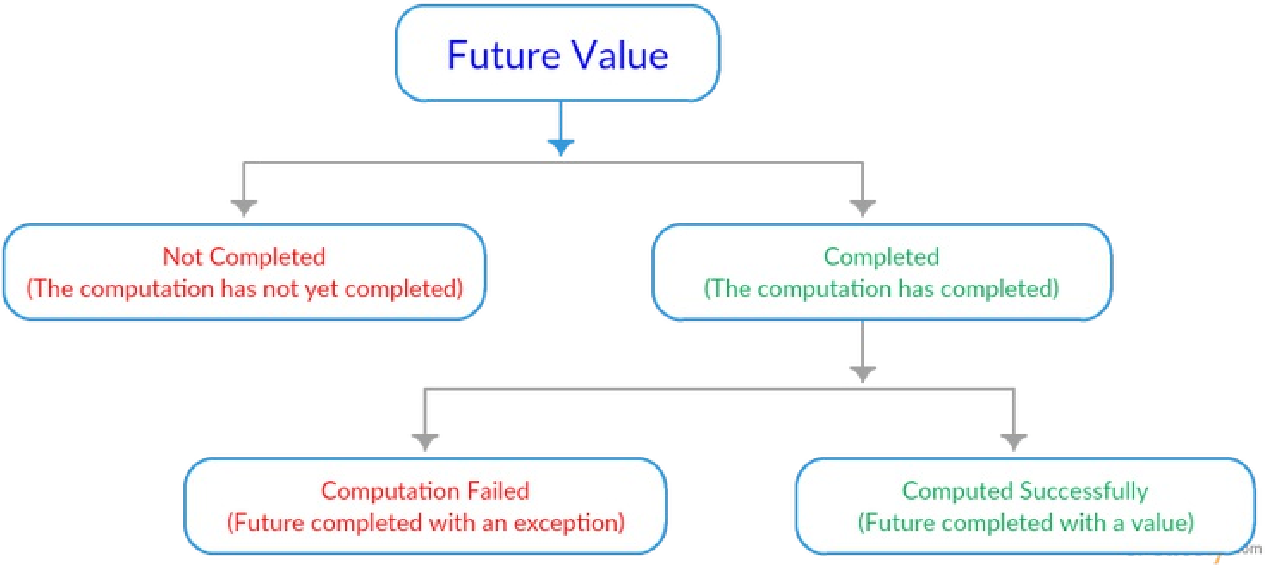
Back to the Scala Future

What is a Future?

A future is a place holder which holds a value that may become available at some point. A future value can be understood by following figure.


Future Value
Not Completed
(The computation has not yet completed)
Completed
(The computation has completed)
Computation Failed
(Future completed with an exception)
Computed Successfully
(Future completed with a value)

A future is a place holder which holds a value that may become available at some point. A future value can be understood by following figure.

The operations which may take time to execute are enclosed under the Future scope which starts asynchronous computation and returns a Future value holding the result of the computation. The execution of the code enclosed in the future block depends on the execution context associated with the block. The execution context is the environment in which the code executes; it is further associated with a thread pool. The thread pool provides the threads on which the Future enclosed block of code is executed. Hence whenever code enclosed in Future block is encountered a new thread from the thread pool is picked for its execution thus performing many operations in parallel. We generally use ‘The Global Execution Context’ which is associated with ‘ForkJoinPool’.

```
import scala.concurrent.Future     //library to use Future
import scala.concurrent.ExecutionContext.Implicits.global     //makes implicit
global execution context available

def fetchData(url:String): Future[String] = {

    Future {     //The Future scope/block

        val myWebService = new RandomWebService()
        myWebService.fetchContentFromUrl(url)     //method fetch data from the
specified URL and returns String data (time consuming operation)
    }
}     //Since the call to fetch data is enclosed in future the method will
return Future[String] when the data becomes available
```

Access result of Future

The figure in checkpoint 1 clearly shows the two probable results of the Future i.e. either success or failure. The result (value or exception) from a Future can be accessed by following ways

- Callbacks
- Combinators
- For Comprehensive
- Await.result
- The async library

Callbacks

Result of a Future can be accessed by registering a callback for the Future. This callback is called asynchronously once the future is completed. A callback needs an execution context since it is also executed asynchronously. Callback can be registered using following methods

- onComplete

The onComplete Method allows handling of both failed and successful future computations. Following example explains its use case

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

val result:Future[String] = fetchData("www.RandomURL.com")     //The method defined above, it returns Future value

result.onComplete{     //Executed asynchronously when the future is completed

    case Success(data) => println("the fetched data from the web site is - "+data)     //case is executed if a value is returned by the future without any exception
    case Failure(ex) => println("ERROR occured - "+ex.getMessage)     //This case is executed in case of exception
}
```

The above code clearly explains how the value returned by a future is accessed in case of a success or a failure.

- onSuccess

The onSuccess Method allows handling of only successful future computations. This callback is executed only when a future returns data otherwise its not executed. Following example explains its use case

```
val result:Future[String] = fetchData("www.RandomURL.com")     //The method
defined above, it returns Future value

result.onSuccess{     //Executed asyncronously only when the future is
completed with a SUCCESS otherwise skipped

    case data => println("the fetched data from the web site is - "+data)
//This case is executed only if a value is returned by the future without any
exception
}
```

- onFailure

The onFailure Method allows handling of only failed future computations. This callback is executed only when a future results in exception otherwise its not executed. Following example explains its use case

```
val result:Future[String] = fetchData("www.RandomURL.com")     //The method
defined above, it returns Future value

result.onFailure{     //Executed asyncronously only when the future is
completed with an exception(FAILURE) otherwise skipped

    case ex => println("ERROR occured - "+ex.getMessage)     //This case is
executed only in case of failure and has handler for exception.
}
```

Combinators

Combinators are another way to access the value of a future. The combinators act on a future value and returns corresponding new future. Flatmap, map, filter, recover,recoverWith,fallBackTo are some of the examples of the combinator. One thing to note is that the combinators are internally implemented using callbacks. Combinators are used when we want the result of a Future variable/constant and perform computations on it and return the result of the computation as a new future. Following example demonstrate the usage of the combinators (map)

```
val futureResult:Future[Int] = Future{giveMeANumber()}     //Let the method
'giveMeANumber' is time consuming and returns an integer value

val incResultByFive :Future[Int] = futureResult.map(x => x+5)     //using map
we access the result value of the future variable 'futureResult' and perform
operations on it.
```

There is a problem in above code, if the result in the Future ‘futureResult’ is a failure, it yields an exception. Hence its a good practice to provide some kind of handler to handle the exception which may occur while using the futures. Recover, recoverWith, fallbackTo are some ways by which the exception generated by the future can be handled. The recover combinator creates a new future which holds the same result as the original future if original future completes successfully otherwise it handles the exception and provides a default value which acts as the resultant Future. Following examples explains the concept

```
val futureResult:Future[Int] =Future{giveMeANumber()}     //Let the method
'giveMeANumber' is time consuming and throws an exception

val incResultByFive :Future[Int] = futureResult.map(x => x+5).recover{case ex
=> -1}     //Here we try to use value of 'futureResult' which is an exception
hence the control flows to 'recover' block and 'incResultByFive' is assigned
the default value of Future(-1).
```

For Comprehensive

Combinators are a very efficient mechanism to access and use the value of a Future but they have a limitation i.e. it becomes a bit complex to use combinators when we have to use several futures to compute a result. For example we have to add two future values and return a new future as there sum. For comprehensive is used for situations like this when we have to use multiple future values to compute the result. Following example will explain there usage better

```
val first:Future[Int] = Future{
longRunningTask1()
}

val second:Future[Int] = Future{
longRunningTask2()
}

val addResult:Future[Int] = for{     //the for comprehensive makes the future
values available to be operated
one <- first     //here one is of of type int
two <- second
} yield one + two     //yield returns the Future of value computed by adding
two integers

val finalVerifiedResult:Future[Int] = addResult.recover{case ex:Exception =>
-1}     //assigns a value of -1 to final result if the 'addFuture' yields
Failure ie Exception.
```

In above code the ‘addResult’ future is completed only when ‘first’ and ‘second’ future are completed. The ‘finalVerifiedResult’ gurentees that the final future value will always be a Success and will have the default value of -1 in case of Failure of ‘addResult’.

Await.result

Await is an object available in ‘scala.concurent’. It has two methods

- ‘ready’. It waits for the “completed” state of an Awaitable(Future[T])
- ‘result’. It waits and returns the result (of type T) of an Awaitable(Future[T])

Internally Await use blocking and with the ‘result()’ method it yields the result of associated Future by blocking the current thread and thus killing the asynchronous approach to code. Hence, It must be proffered to use Await only in test cases, to test the functionality of asynchronous code. Following example demonstrate the usage of Await

```scala
import scala.concurrent.Await
import scala.concurrent.duration._

val myFuture:Future[Long] = Future{computeFactorial()}
//'computeFactorial()' method returns a long.

val actualResult: Long = Await.result(myFuture, 10 seconds)     //here we use the 'result' method, 'myFuture' is the future in whoes vales we are intreasted in.
```

References

[Futures on docs.scala-lang (http://docs.scala-lang.org/overviews/core/futures.html)](http://docs.scala-lang.org/overviews/core/futures.html)