# Greetings - Scala fundamentals (part 8)

## Aims

We will be covering the following topics:

- Exception Handling
- Try
- Either
- Recursion
- Tail recursion

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 Greetings - Scala fundamentals (part 7).
You have implemented your solution for `PetRepository`.

## Lesson steps

In *Greetings - Scala fundamentals (part 7)* we got you to work in groups to implement `PetRepository`.
We are now going to instantiate the `PetRepository`, `add` Pets to our repository, and call the other methods.

An example implementation of `PetRepostiory` is PetRepository example on GitHub.

This demonstrates how to invoke the methods of `PetRepository`, for example, how to `def add(pet : Pet*)` add multiple pets.

> Did you attempt to call the `update` method passing in a `Pet` returned from `findByName()`?
>
> Did you notice how this was extremely difficult as both of `Dog` and `Cat` extended the

> generic trait `Pet`. This was largely due to not being able to `.copy()` and instance of a Pet.
>
> Why do you think this is?

An example of how we have tested the implementation is Testing PetRepository GitHub

## Exception handling

We have encountered and discussed the concept of an `Exception` a couple of times already while building our application. Can you remember any of the instances when we encountered an `Exception` and what caused it?

- In part 2, we tried to input the `String` "X" where our code expected an `Int` and encountered a `NumberFormatException`.
- In part 7, we discussed the `NullPointerException` when we learned about `Options`, `None` and `null`.
- In part 7, we also encountered the `MatchError` during our discussion of **pattern matching**.

In our pattern matching example, we used the `_` operator to avoid getting a `MatchError`:

```
val number = 1

def intToString(n : Int) : String = {
    n match {
        case 0 => "zero"
        case 1 => "one"
        case 2 => "two"
        case _ => "above two"
    }
}
```

Our `case _ => "above two"` statement ensures that we can provide a match for any possible value of `Int` that might be passed to `intToString()`.

In the rest of our code, however, we've haven't taken any steps to deal with the possibility that we might encounter an exception. An exception occurs as a result of some kind of failure in our application. This might be caused by:

- something we *could* expect or that is internal to our code, such as a user inputting the value "one" instead of 1 when asked for their age, **or**
- something that we *couldn't* expect or that is external to our code, like a database or

server that we depend on being unavailable, **or**
- something that is a *fatal* exception, like an `OutOfMemoryException`

Generally, we should assume that where a problem could occur, it will, and write our code to cover these *exceptional* scenarios as gracefully as possible. Exceptions are useful for developers, as they help us understand where and when a problem has occurred, but not so much for users of our application. In the principles of functional programming, we strive for pure functions therefore should not have any **side effects** or exceptions.

> An example of a side effect is logging to the console, throwing an exception or calling an API. Anything other than the expected return value.
> In programming, we obviously need some side effects to do things like render to a screen, therefore we tend to encapsulate our side effects in the outer layer of our application.

**Further reading**

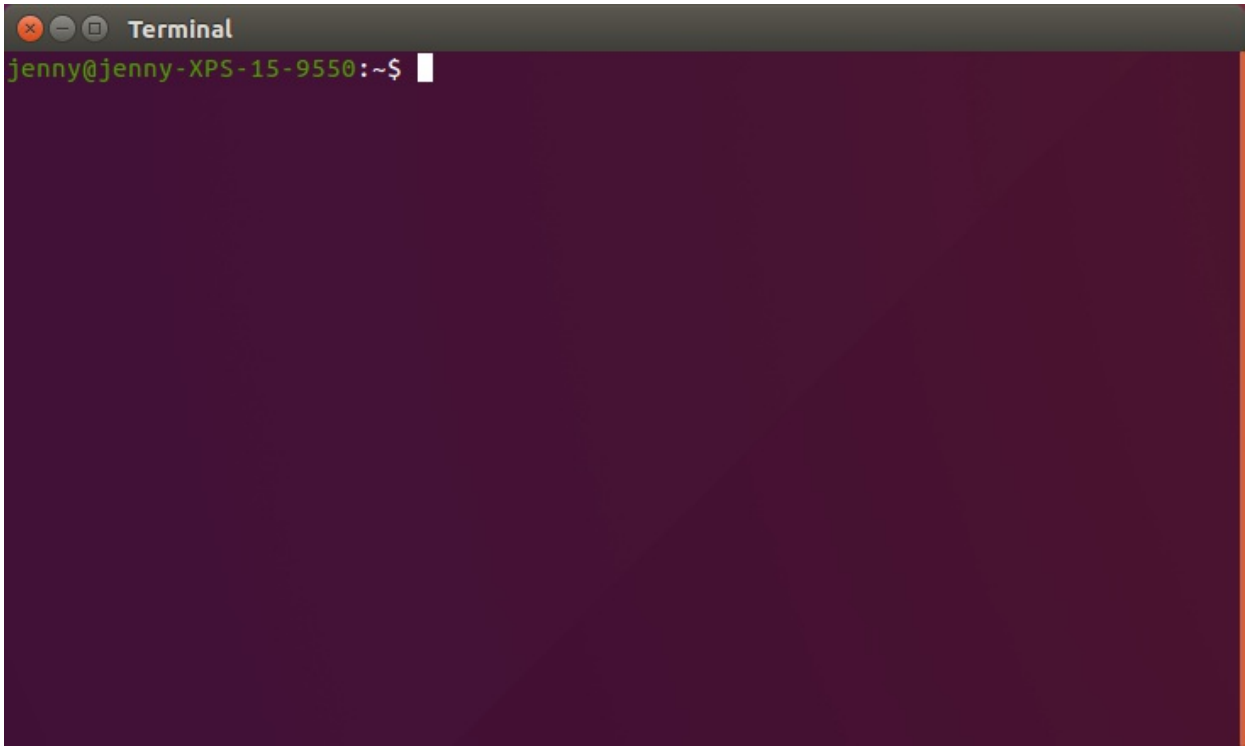
So You Want to be a Functional Programmer (Part 1)

**Task**

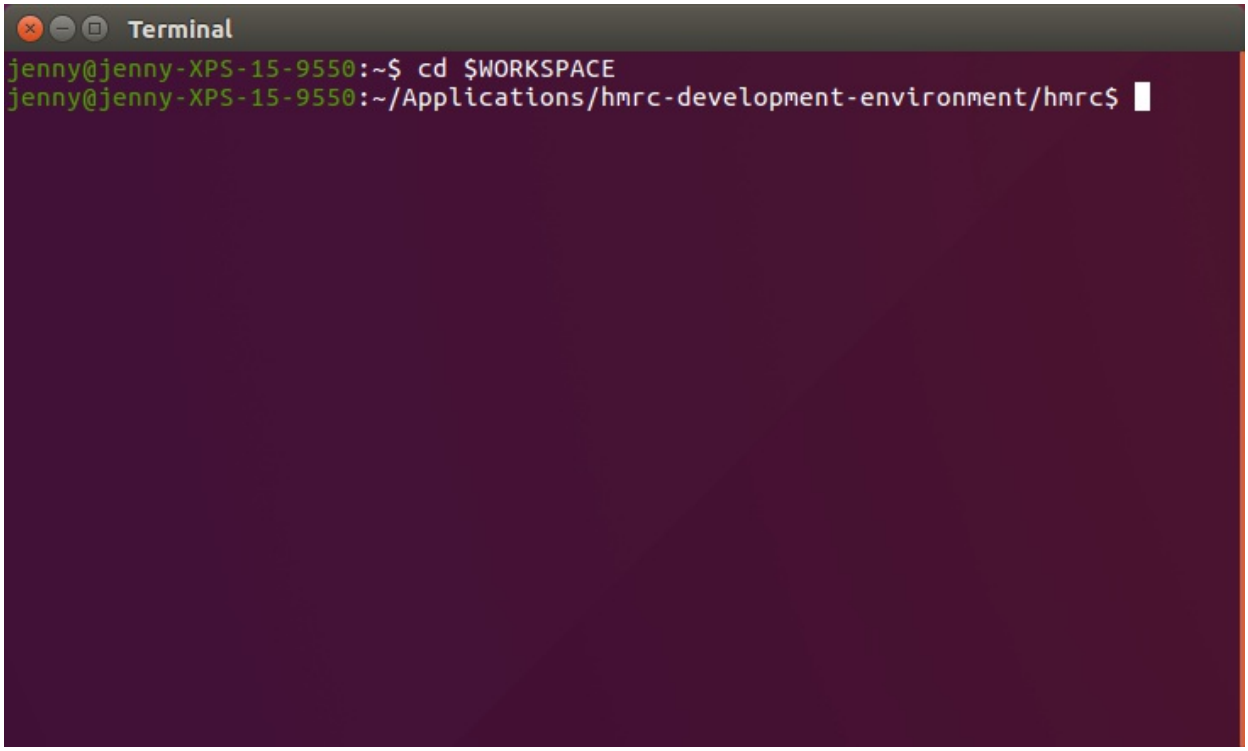
Let's investigate how we can handle the `NumberFormatException` caused by our `GreeterApplication`.

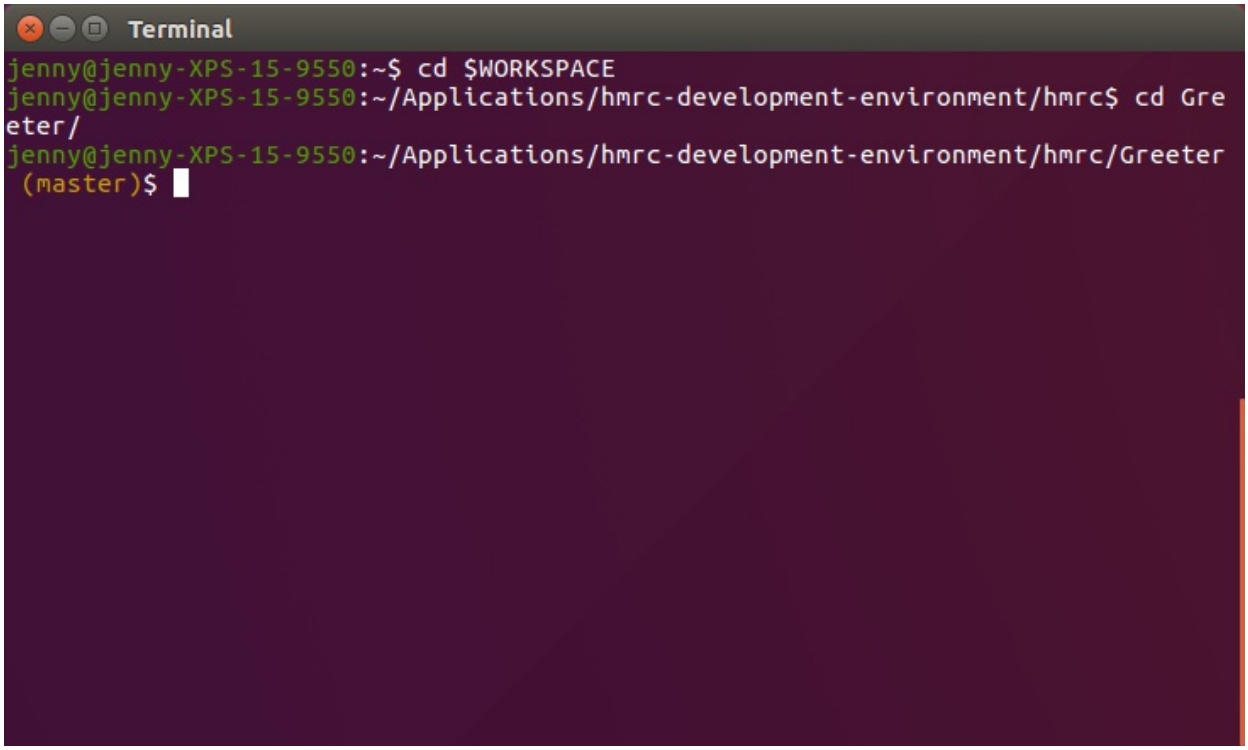
Up to now, we've been running our application using IntelliJ. Now, we're going to try running it from the command line, using the `sbt` command.

First, we need to open a terminal window.

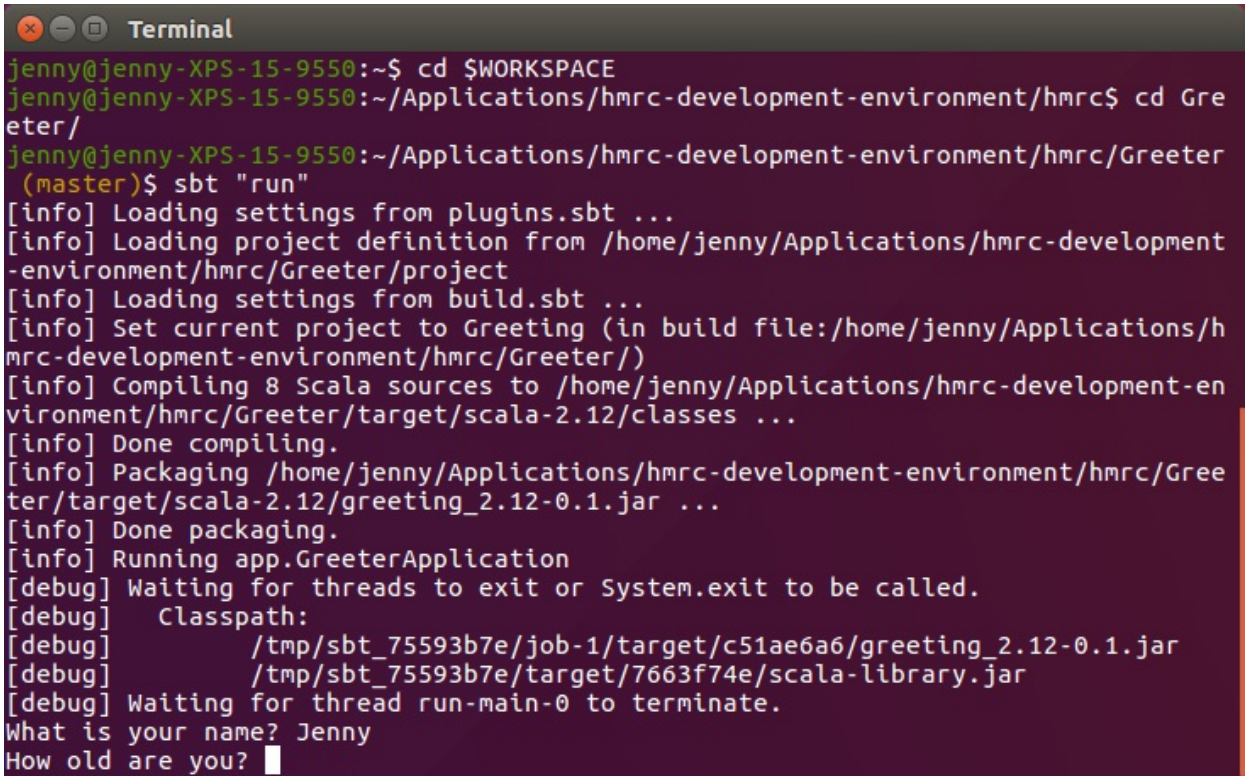



Next, we enter `cd $WORKSPACE` then `cd Greeter` to navigate to our work area.






Now that we're in our project folder, we can run our application with the command `sbt run`.




We have now successfully run our application using the terminal.

If we run our application and enter "x" when prompted for an age, we'll see the following:

```
Exception in thread "main" java.lang.NumberFormatException: For input string: "x"
    at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65
    at java.lang.Integer.parseInt(Integer.java:580)
    at java.lang.Integer.parseInt(Integer.java:615)
    at scala.collection.immutable.StringLike.toInt(StringLike.scala:301)
    at scala.collection.immutable.StringLike.toInt$(StringLike.scala:301)
    at scala.collection.immutable.StringOps.toInt(StringOps.scala:29)
    at app.GreeterApplication$.delayedEndpoint
  $app$GreeterApplication$1(GreeterApplication.scala:14)
    at app.GreeterApplication$delayedInit$body.apply(GreeterApplication.scala:6)
```

We can use the stack trace of this `NumberFormatException` to identify where in our code we need to make changes. Each line refers to a specific line of code, either in our application or in packages that we are making use of. If we work from the bottom up, we can trace the exception through to it's point of origin.

At the very bottom of our stack trace, we can see the following:

```
at app.GreeterApplication.main(GreeterApplication.scala)
```

This confirms that running our application has caused the problem. As we move up the stack trace, we can see what might have caused the failure:

```
at app.GreeterApplication$.delayedEndpoint$app$GreeterApplication$1(
  GreeterApplication.scala:14)
```

If we open our `GreeterApplication` code at the relevant line - the number after `GreeterApplication.scala:` - we should see:

```
val person = new Person(name, age.toInt, List(withdrawn, deposited))
```

Here, we're calling the `toInt()` method on age. This is confirmed as the source of the problem higher up the stack trace:

```
at java.lang.Integer.parseInt(Integer.java:580)
```

The application has attempted to call the `parseInt()` method in `java.lang.Integer` on our `age` argument. It's not possible to convert our input - "x" - into an `Int`, so the exception occurred.

So, what can we do about this?

**Task** *1 Hour, group exercise*

Refactor our `GreeterApplcation` so that it will no longer cause a `NumberFormatException` if we enter "x" as our age in response to the prompt "How old are you?"

The user should be prompted again for their age, ideally with a message that explains they need to enter an actual number for their age. Work in groups to solve this problem, and then present your solution back to the rest of the academy.

**Further reading**
Scala Exception Handling

**Tip**

> To solve this problem, we will need to refactor our code so that we encapsulate the exception into it's own method, and make a recursive call to collect the relevant arguments again.

> When making recursive calls, be careful not to "blow the stack". This will eventually result in a StackOverflowError.

**Further reading**
What is a StackOverflowError?

Example implementation that would handle the exception thrown by `toInt`.

ExceptionHandling the toInt method - GitHub

In our example implementation, we've handled a single type of exception. If we had an operation that could result in more than one exception, we could introduce additional case statements to the `catch` block of our code. If we encountered a `NullPointerException` exception in our example above, for example, this would not be caught by our `catch` block.

> There is an additional clause we can add to our `try` `catch` block: `finally`. If we know that we want a piece of code to execute regardless of how the previous blocks evaluate, we can use `finally`

```
try {
    {...}
} catch {
    {...}
} finally {
    {...}
}
```

> The code in the `finally` block will execute, regardless of the outcome of the `try` and `catch` blocks.

We'll be refactoring our `arguments()` function later. For now, let's have a look at an example of another way to handle an exception. We can use methods provided by the `scala.util.control.Exception._` package. An example of this, using the `allCatch()` function, is below:

```
val n1 = Exception.allCatch.opt(3)
val n2 = Exception.allCatch.opt(0/0)
```

Normally, `0/0` would produce an `ArithmeticException`, but we're going to catch it. As the last function on this chain - named `opt()` - suggests, we're actually going to get an `Option` from both `n1` and `n2`.

```
val n1 = Exception.allCatch.opt(3) // returns Some(3)
val n2 = Exception.allCatch.opt(0/0) // returns None
```

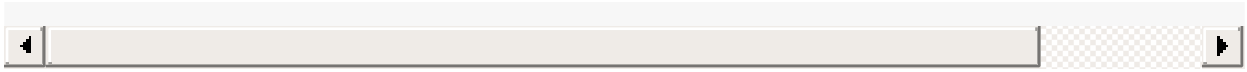
In the case of `n2`, the exception is caught and becomes a `None`. In the case of our `arguments()` method, this wouldn't completely solve our problem.

```
val age = Exception.allCatch.opt("hello")
```

In this case, we would get `Some("hello")` as an `Option[String]`.

You can also use inheritance to implement your own exceptions. This is useful to provide further context within your code. I.e. there may be an instance where an `Exception` may not be explicit enough.

```
def findByName(name : String) : Option[Pet] = _pets.find(p => p.name.equalsIgnoreCa
```



The example above will return an Option which will resolve to either a `Some` or `None`.

We *may* want to provide further context and throw an exception instead, especially if this is a *side effect* that we cannot control.

```
object NoPetFoundException extends Exception

def findByName(name : String) : Pet = {
    val found = _pets.find(p => p.name.equalsIgnoreCase(name))
    found match {
        case Some(p) => p
        case None =>
            // imagine that this is a side effect that should rarely happen
            throw NoPetFoundException
    }
}
```

The example above provides further context to a checked exception, and this can be handled at the most outer layer of our application.

**Further reading**
scala.util.control.Exception

## Try

Scala has another data type known as `Try`. `Try` is similar to that of `Option[T]` in that it has two concrete implementations that you will come across. These are `Success[T]` and `Failure[T]`.

> You can use `Try` instead of *try/catch* block to postpone exception handling.
> Success[T] wraps the result value of type T, this could be a Pet for example: Success[Pet].
> Failure wraps an `Exception`, a Failure can only contain a type of `Throwable`.

**Further reading**

What is Java Throwable

Let's look at how this can be used to handle an exception instead of try/catch:

```
# try/catch approach

val age : String = "adam"

try {
    age.toInt
} catch {
    case _ : NumberFormatException =>
        println("Please enter a number for your age")
}

# Try approach

def parseAge(age : Any) = Try(age.toInt)

parseAge("adam") match {
    case Success(age) =>
        println(s"You are $age years old")
    case Failure(exception) =>
        println("Please enter a number for your age")
}
```

We can also use the same methods that are available on `List`, `Option`:

```
parseAge("adam").map(age => s"You are age years old")
```

We can also iterate over a `Try` and `Option`:

```
for {
    age <- parseAge(inputAge)
} yield s"Your age is $age"
```

This will extract the value from the value returned from the `Try[T]`. If this successfully parses then it returns a `Success(12)` into the *variable* age. If an exception is thrown this will return a `Failure[T]`.

In the example above, `map()` will only be called if the `Try[T]` resolves successfully. It will fail silently and return a `Failure[Exception]` back instead. You can then safely handle this exception as it does not change the return type of the function/method.

**Tip**

> Only non-fatal exceptions will be caught by the Try, fatal exceptions such as

> OutOfMemoryException will still be thrown::

You can also determine how the `Try[T]` resolved by using either `isSuccess()` or `isFailure()`.

**Further reading**
Try, Option or Either

Let's refactor our solution to use a `scala.Try` instead of implementing a *try/catch* block.

We're going to return to the `arguments()` method in our `GreeterApplication`, to introduce `Try` into our code. In our examples above, we've seen how we might go above refactoring from a `try catch` block to using a `Try`. We now want to implement this, so that when our `GreeterApplication` is run and captures our age, we'll be re-prompted for input if we have been unable to parse the provided age into an `Int`.

**Task** *15 minutes, individual exercise*

> Use the examples we've covered to refactor your code for the method `arguments()` to use `Try`.

Discuss with the rest of the class how this next implementation is different. Is it better, worse? Why?
Use the further reading material to help you explain why.

> Don't forget to commit your code.

# Either

An `Either` is another data type in Scala. Unlike other types, an `Either` actually represents two different types. It can look something like:

```
def toIntEither(s: String): Either[String, Int]
```

Can you remember where in our code or in the academy material we have seen a declaration similar to this before?

Our `arguments()` function has a return type of `(String, Int)`. We've also encountered the `Tuple`, which can contain more than one data type. As the name of the `Either` suggests, however, it represents *either* one thing, or the other. Whilst our `arguments()` method returns both a `String` *and* an `Int`, our `toIntEither()` function above will only return one or the

other.

An `Either` can be used as an alternative for an `Option`. Instead of returning a `Some` or a `None`, the `Either` returns a `Right` or a `Left`. The `Left` takes the place of the `None`. Unlike a `None` however, the `Left` can contain useful information. `Right` takes the place of `Some`.

In our example above, the `Left` is a `String` and the `Right` is an `Int`. We might expand this function, so that it tries to parse `s` into an `Int`. If this is possible, it would return a `Right()` with the new `Int` value inside it, e.g. `Right(3)`. If it's not possible to parse `s` as an `Int`, it would return a `Left()` containing a string, e.g. `Left("Error: you didn't enter a number")`.

```
def toIntEither(s: String): Either[String, Int] = {
    try {
        Right(s.toInt)
    } catch {
        case e: Exception => Left("Error: you didn't enter a number")
    }
}
```

Like other container types such as `Option`, we have access to methods that allow us to check the contents of an `Either`: `isLeft` and `isRight` do what their names suggest, returning a `Boolean` depending on the contents of the `Either`.

> Where `Either` differs from `Try` is that it is not *success* based. A `Try` is either a `Success` or `Failure`, but an `Either` is technically unbiased. Conventionally, however, we tend to treat `Left` as a failure.

We can also call `.left` or `.right` on our `Either`. This doesn't get us the contents of the `Either`, however. Instead, calling `.left` or `.right` provides us a `LeftProjection` or `RightProjection`. We can then operate on these projections, although - crucially - the operation will only be performed if the value within the `Either` matches the projection type we've specified. `.left` will operate only on a `LeftProjection` and `.right` on a `RightProjection`.

```
val x = toIntEither("wow").right.map(z => z * 2)
val y = toIntEither("3").right.map(z => z * 2)
```

Consider the above. What do you think `x` and `y` would return?

In both cases, the type of `x` and `y` is `Either[String, Int]`.

Let's step through the process of evaluating each:

**val x**

1. `toIntEither` is called with "wow" as the argument
2. It tries to parse "wow" as an Int, which results in an Exception
3. The Exception is caught, and a `Left` is returned
4. We then call `.right` and get a `RightProjection`
5. As the `Either` contains a `Left`, the value inside it remains untouched
6. We return `Left("Error: you didn't enter a number")`

**val y**

1. `toIntEither` is called with "3" as the argument
2. It tries to parse "3" as an Int, which returns the Int value 3
3. We then call `.right` and get a `RightProjection`
4. As the `Either` contains a `Right`, the value is multiplied by 2
5. We return a `Right(6)`

If we had instead called `.left` then `y` would have return `Right(3)` and `x` would have return `Left(Error: you didn't enter a numberError: you didn't enter a number)`. In this case, the `z => z * 2` transformation has only been applied to the `Left` value, not the `Right`.

**Further reading**
scala.util.Either

**TDLR; a summary of Option, Either and Try**

1. `Option[T]`, use it when a value can be absent or some validation can fail and you don't care about the exact cause. Typically in data retrieval and validation logic.
2. `Either[L,R]`, similar use case as Option but when you do need to provide some information about the error.
3. `Try[T]`, use when something Exceptional can happen that you cannot handle in the function. This, in general, excludes validation logic and data retrieval failures but can be used to report unexpected failures.
   Exceptions, use only as a last resort. When catching exceptions use the facility methods Scala provides and never catch `{ _ => }`, instead use `catch { NonFatal(_) => }`

# Recursion

You have come across recursion already in the material that has been introduced.

Recursive methods can be thought of in the same manner as we have when using `for loops` over a collection. i.e. we want to perform an expression on a collection of data items.
We can achieve this by implementing functions that make a recursive call to themselves. This will evaluate an expression over each item in a collection.

They are also useful if you want to achieve *flow control* like we did in the `arguments()` example. This allows you to define logic based on conditions in your code. i.e. ask for data multiple times until satisfied.

Let's look at an example of a recursive method that will sum up the total of a `List[Int]`.

```
def sum(ints: List[Int]): Int = ints match {
  case Nil => 0
  case x :: tail => x + sum(tail)
}
```

In the example above, we have declared a method that takes a `List[Int]` as it's argument. We then perform a **pattern match** on the list, providing two possible cases.

The first case `case Nil`, signifies that if `ints` is an empty list `List[Nothing]` then go into this expression and return `0` as the result.

The second case `case x :: tail` is known as an **Extractor** that is provided by the `List` class. This provides an `unapply()` method which returns a **Head** and a **Tail**. Head in this instance is a single element of the list, and the tail is the list minus the head. i.e. `List[Person]`, head is `Person`, tail is `List[Person]` *if there is more than one element, otherwise it would be* `Nil`.

If it matches the second case, it will evaluate the expression and add the result to the *stack*, in short, a value is held in memory to be used again later.

**Further reading**

https://en.wikipedia.org/wiki/Stack_(abstract_data_type)

A recursive call holds all of the evaluated expressions in the stack, once it reaches the bottom of the collection, it then traverses back up the collection to total all the values.
If the input `ints` is a long list with 100+ numbers. Then this will eventually throw a `StackOverflowError` exception as there is no memory left to allocate the value of the expressions.

## Tail recursion

In the `sum` example above we could calculate the total of a `List[Int]` using a recursive implementation. However, knew this would blow the stack if the input was a large list of integers.

We can solve this in Scala using a tail recursive implementation.

**What does tail recursion mean?**

Tail recursion is where we have implemented a recursive solution, however, the last thing the implementation does, is to call itself with the current result passed through as an argument, or runs an exit expression.
Scala implements this and we can ensure a function is tail recursive by using an *annotation* which can be imported.

```
package calculator

object Calculator {

    import scala.annotation.tailrec

    def sum(nums : List[Int]) = {

        @tailrec
        def sumHelper(ints : List[Int], acc: Int) = {
            ints match {
        case Nil => accu
        case x :: tail => sumHelper(tail, accu + x)
      }
        }

        sumHelper(nums, 0)
    }
}
```

The example above, implements a `sum` method which accepts our argument we want to calculate the sum of. We have declared an inner function called `sumHelper` which we invoke as the last expression that `sum` performs. Passing in `nums` as our `List[Num]`, and `0` as our current *accumulator*.
In short, an *accumulator* is our current running total.

Inside of our `sumHelper` method we have first pattern matched our list, if the list is empty `Nil`, we run an exit expression and return the current *accumulator*. This may be `0` or a value.

If the list is not empty, we then use an **Extractor** `unapply()` to return the `head` and `tail` of the list. We then make a call to our helper function again passing though the remainder of the list, and the value of `0 + number`.

An example of this is:

```
val input = List(1, 2, 3)

val sum = sum(input) // returns 6
```

This has calculated `6` by doing the following:

1. Iteration 1, evaluates `0 + 1` as `1`, passes `1` through
2. Iteration 2, evaluates `1 + 2` as `3`, passes 3 though
3. Iteration 3, evaluates `3 + 3` as `6`, passes 6 though with tail being an empty list
4. Iteration 4, `ints` is `List[Nothing]` therefore returns `6`

We have ensured the method is tail recursive by using the `@tailrec` annotation above our helper method.

**Task** *30 minutes, paired exercise*

> Implement a tail recursive function to sum a total value of `Ints`. The input will be `List[Int]`.
>
> Add `println` statements so that we can see what the current value of the *accumulator* and *tail* is at each iteration.

Don't forget to commit your code.

# Recap

In this session, we have covered a number of new topics.

- Exception Handling
- Try
- Either
- Recursion
- Tail recursion

In learning about these topics, we did the following:

- Worked through a stack trace, to understand what had caused an exception
- Refactored a number of the methods in our `GreetingApplication` to use these new concepts.
- Started to run our application using the terminal, rather than through IntelliJ.
- Understood the difference between `Try`, `Option`, and `Either`
- Understood how to implement a recursive function to traverse a collection
- Implemented a tail recursive function and understood it's usage of the stack

In the next part, we will be learning about unit tests and implementing them as part of our `GreeterApplcation`.

# Resources

- Scalacheat | Scala Documentation
- So You Want to be a Functional Programmer (Part 1)
- Scala Exception Handling
- What is a StackOverflowError?
- What is Java Throwable
- scala.util.control.Exception
- Try, Option or Either
- scala.util.Either