Functional programming fundamentals

- Theoretical part
  - What is functional programming
  - Why functional programming matters `15 min`
- DIY part
  - Basic Scala `10 min`
  - Algebraïc data types `10 min`
  - Pattern matching `10 min`
  - Recursion `10 min`
  - Typeclass `15 min`

Introduction

Welcome to the workshop Functional programming, where you will get in touch with the absolute basics of functional programming. This workshop consists of several parts. First we will introduce explain what functional programming. Then we will take a closer look at several reasons why functional programming is increasing in popularity. This will be followed by a hands-on, where you will begin your journey through the basics of Scala and FP. You will play around with algebraïc data types, pattern matching, recursion and typeclasses. All topics will be explained and there are exercises in between that requires you to put the gained knowledge into practice.

Let's get started right away!

What is functional programming

As you would have expected, functions are the primary building block of application written in FP. This is in contract to object-oriënted programming, where classes are used to construct programs. Functions are so called "first class citizens", which means that functions are like data. You can just use them as any other ordanairy type like String and Int. You can for example pass them as a parameter to another function or have a function that returns another function when called. A function that either receives atleast one parameter as argument or returns a function is called a higher-order function.

A function in functional programming are expressions, which is in contrast to imperative programming language where they consist of a number of statements which will be executed sequentially. A definition given by Wikipedia: “Expressions are a combination of one or more explicit values, constants, variables, operators and functions that the programming language interprets and computes to produce another value". There are slightly different flavors of functional programming. We will pay attention to purely functional programming. This will restrict functions in a sense that they need to be pure. What does it mean to write a pure function?

A function is pure if it is referentially transparant. "An expression e is referentially transparant if, for all programs p, all occurrences of e in p can be replaced by the result of evaluating e without affecting the meaning of p. A function f is pure if the expression f(x) is referentially transparant for all referentially transparant x.”. Read that sentence a couple of time and make sure you understand. Simply said, we can substitute the function with it's predicted result without altering the behaviour of the program. The following two functions are not referentially transparant:

```
// This function is not referentially transparant, because if we substitute
// the function call to function1() with "hello world", we wouldn't have a println,
def function1() = {
  println("test")

  "hello world" //scala uses implicit return where the last statement will be retur
}

// This function is not referentially transparant, because the output is unpredicta
def randomNumber() = {
  Math.random ....
}
```

There are multiple ways to break referential transparancy. Everything that breaks it is called a side-effect. In the previous examples, println("test") and Math.random are the side-effect and the unpredictable parts of the function.

The following function is referentially transparant

```
// This function is referentially transparant because we can substitute every funct
// If the function call is identity(3), we can replace it with 3 without changing t
def identity(a: Int): Int = a // returns it's input argument
```

Having functions that are referentially transparant provide some great benefits. Reasoning about software becomes easier and software is more likely to be correct. We will discuss some advantages of functional programming in a bit. First we are going to take a look at why having referentially transparant functions is important.

This example originally comes from the book Functional Programming in Scala and shows that functions that contain side-effects do not compose. Consider the following function written in Scala. More details will be explained later, but the intend should be clear. This function will give you a coffee given a CreditCard and a PaymentService. There is an obvious side-effect in the function, which is calling the PaymentService. This will charge the CreditCard for the price of a cup of coffee.

```
class Cafe {
  def buyCoffee(cc: CreditCard, p: PaymentService): Coffee = {
    val cup = new Coffee()
    p.charge(cc, cup.price)

    cup
  }
}
```

We can show that functions that contain side-effects do not compose by creating a function that buys an arbitrary n cups of coffee. A function like this can be implemented as following.

```
class Cafe {
  def buyCoffees(n: Int, cc: CreditCard, p: PaymentService): List[Coffee] = {
    var coffees: List[Coffee] = List()

    for (a <- 1 to n) {
      // prepend the new coffee to the list
      coffees = buyCoffee(cc, p) :: coffees
    }
  }
}
```

While this implementation is technically correct, it's most likely not the result that we want to achieve. The PaymentService will be charged n times and will return n coffees. Also other problems will arise with this implementation. Assume that the charge method on the PaymentService will send a request that has a chance to fail. What do we do when 1 call to the PaymentService fails?

Functional programming to the rescue. We solve this by making the effect explicit in the type signature. In functional programming, we could implement this as following

```
class Cafe {
  def buyCoffee(cc: CreditCard): (Coffee, Charge) = {
    val cup = new Coffee()
    (cup, Charge(cc, cup.price)
  }

  def buyCoffees(cc: CreditCard, n: Int) : (List[Coffee], Charge) = {
    val purchases: List[(Coffe, Charge)] = List.fill(n)(buyCoffee(cc))

    // unzip transforms a list of tuples to tuples of two lists.
    val (coffees, charges) = purchases.unzip
    // We combine the charges by reducing n charges to one charge by accumulating t
    (coffees, charges.reduce((c1,c2) => Charge(c1.cc, c1.amount + c2.amount)))
  }
}

class PaymentService {
  // The IO[Unit] can be read as: When I perform, I'm likely to perform some form o
  def charge(charge: Charge): IO[Unit] = ..... // pay the bill
}
```

We obviously still have the side-effect, but we gained several things by encoding it like this.

- By splitting the side-effect from the business logic, we achieved a piece of code that is very easy to test and is deterministic. We can test buyCoffees by just providing a creditcard and the number of coffee we want and we can verify this by just inspecting the output. This is in contract with side-effectful functions/classes, where functions and classes will depend on the dependencies which can be mocks or stubs in test. This way you basically transforms a non-deterministic function to a deterministic function. Also, our code is nothing more than transforming data which is easier to reason about.
- Our function is now reusable as shown in the example. We can compose our functions since it is doing nothing more than the signature actual tells us.

The next chapter will introduce a couple of benefits of functional programming.

Why functional programming matters

Before we can start using functional programming, it's always useful to look at the pre-claimed benefits and what goal it's trying to achieve. In case of functional programming there are several reasons people claim it's great and they explain the major reasons of the increase in popularity. Three reasons will be described briefly below.

Concurrency & parallelism

The importance of concurrency and parallelism is increasing. Gorden Moore, co-founder of Intel predicted in 1965 that the number of transistors on processors would double every year.

A couple of years later he adjusted his claim to a doubling every two years. This claim is called Moores Law.

Neil Thompson gave a populair rephrasing of Moores law: "Computing performance doubles every couple of years.". In figure 1 it's easy to see that the number of transistors is still doubling, but the effects positive effects are stagnating.

For that reason, instead of putting more transistors onto a CPU, computers will get multiple CPUs (cores). This means that the time of free performance increases are gone and it's time that the cores are utilised as intended. This means an increase in importance of concurrency and parallelism.

Concurrency and parallelism is very difficult. When using an imperitive programming language, a common this to do is to create variables with values that can change of time. This contains some problems. Look at the example below:

```
public int x = 1;

void function1() {
  int v1 = x;        // 1
  int v2 = x;        // 2

  x = v1 + v2 + x;   // 3
}

void function2() {
  x = x * 5;         // 4
}
```

The problem is that in a multiprocessor environment it is impossible to reason about this code. Let's assume that the lines are executed in the following order: 1, 2, 3, 4. This means that the result will be 15. But in an multiprocessor environment the lines could also be ordered as following: 1, 2, 4, 3. The result in this case will be 7. This problem is called the read-modify-write problem.

When using functional languages where functions are referential transparant, these problems won't exist, since functions are uneffected by side-effects as time, global variables etc.

Basic Scala

Before you are going to practice some basic functional concepts, it's first time to improve your basic Scala.

Paradigms

Scala is a language that fully-supports object-oriented as well as functional programming. Some functional constructs make use of some features of object-oriented and visa versa. This becomes clear when looking at the ADT, which is modelled using subtyping. More information about ADT will come later.

Classes / Objects / Case Classes / Traits

In Java, there are two types of construct you can use to build up your application. These are interfaces and classes. Scala supports 4 different types and they all have their specific use cases.

Classes

Classes are very similar as in Java. A class is a correlation between its data and methods that operate on this data. Take a look at the following example.

```
class Person(val name: String, val age: Int) {
  def printAge(): Unit = { // this example is not referentially transparant
    println(this.age)
  }
}
```

This class has two properties: a name and an age. Both a public properties and will be accessible on instances of this class. Removing the val keywords in front add private before val will make them private. This class has one method. you can add methods with the def keyword (or val, the difference will be explained later). In this case it takes no parameters and returns Unit. It is good enough for now to read is as void.

Creating an instance of a class is the same as in Java.

```
val person: Person = new Person("Mauro Palsgraaf", 21)
```

Note the explicit type after the variable. Scala has type-inference, which means that it could infer this type by itself. It's a choice to write it explicitly, but I like to do it for clarity.

Although classes are used very often in Java, you wont use them that much when using functional programming. The others are more suitable for the job.

Objects

Objects in Scala are singleton objects. If you ever tried programming languages as Haskell, Elm or Elixir, then you can compare them to modules.

Everything in an object is not correlated to an instance. You can compare them in Java as a class only consisting of static methods.

Case classes

Case classes are very similar as classes, but have some differences. You can define a case class just the same as a class, but with the keyword case in front.

```
case class Person(val name: String, val age: Int)
```

To create an instance of a case class, you can do exactly the same as a class without the new keyword.

```
Person("Mauro Palsgraaf", 21)
```

One of the differences of a case class comparing to regular classes is that case classes are equal by values, not by reference. Look at the following snippet

```
class Person(val name: String, val age: Int)

val person1 = new Person("Mauro Palsgraaf", 21)
val person2 = new Person("Mauro Palsgraaf", 21)

person1 == person2 // this expression will return false

-------------------------------------------------------

case class Person(name: String, age: Int)

val person1 = Person("Mauro Palsgraaf", 21)
val person2 = Person("Mauro Palsgraaf", 21)

person1 == person2 // this expression will return true
```

Traits

A trait is very similar to interfaces in Java.

```
trait JsonSerializable {
  def toJson: Json
}
```

This trait defined one method without a body, which classes that implement this trait needs to provide. Note that this is not the way to implement this when using functional programming. In functional programming, typeclasses will be used to encode functionality for types. Typeclasses will be explained later.

Besides defining abstract methods, Scala offers the same functionality as default methods in Java 8, whereas implementations can be provided for methods in a trait by composing the other methods in the trait.

Generics

A function using the def keyword, classes, case classes and traits can all made generic. Generics are encoded in Scala as following

```
def identity[A](v: A): A
```

In this case, the identity function can take any type as input and returns a value of that type as well. Multiple generics can be specified within the square brackets separated by a comma. The name of the generic doesn't matter, but it's convention to go from A up to Z.

Nothing subtype

It can be neccesary sometime to have a subtype of everything. Most of the time this will be useful when encoding ADT. More about ADT will comes later

Algebraïc data types (ADT)

An ADT is a type where we specify the shape of each of the elements. Lets look at an example by looking at the Option for representing missing values a.k.a Option:

```
sealed trait Bool
case object MyTrue extends Bool
case object MyFalse extends Bool
```

We create a sealed trait Emotion. You can add the sealed keyword if you don't want the trait to be extended from outside the file. Traits are also used a interfaces, where you really want this possibility of having a new class implementing the trait.

We can this ADT Bool. Bool is the so called Type Constructor. When specifying types as parameters of functions or as return values, you will use the name of the Type Constructor.

MyTrue and MyFalse are data constructurs and these are the values that are used in your code. In this case it's an logical disjunctions (or), which specifies that Bool is either a MyTrue or a MyFalse.

We can also create an ADT to represent optionallity in our code:

```
sealed trait Option[+A]
case class Some[A](x: A) extends Option[A]
case object None extends Option[Nothing]
```

Again, we create a sealed ADT, because we don't want this trait to be extended outside this.

[+A] means that this is a generic type, where the A needs to be a type itself, like Option[Int]. It can't for example be Option[List], because List itself is a type-constructor instead of a concrete type. You will need to specify the type of the list which would result in Option[List[Int]] or Option[List[String]]. The + in front means that the value is covariant. It means that if B is a subtype of A then option[B] is a subtype of Option[A].

Exercise

1. Implement an ADT to represent Pets. An Pet can be cat with a name, a fish with a name and a Color or a squid with a name and age. A Color can be Blue, Green or Orange and can be represented as an ADT as well.

Pattern matching

Pattern matching is a construct for checking a value against a pattern. It is comparable to a switch statement in Java-like languages, but more powerful.

As example, we will create a function called not, which will negate the boolean.

```
// note that Bool is the ADT you just created yourself
def not(bool: Bool): Bool = bool match {
    case MyTrue => MyFalse
    case MyFalse => MyTrue
}
```

You can see that we match the bool and give it the different options a bool can be. In this case, it's MyTrue and MyFalse. This is not much different that a switch statement. Let's look at a different example where we use the just created Option ADT for optionallity. We create a function addTwoToOption which adds 2 if the option value is present

```
def addTwoToOption(optionInt: Option[Int]): Option[Int] = optionInt match {
  case Some(v) => Some(v + 2)
  case None => None
}
```

As we can see in the example, it allows us to destruct and get access to values in constrast to OO-languages like Java where you would implement getters. In our example, the variable v is a wild-card, where as the pattern matches, will be filled with the value of v. So if Option[Int] is `Some(5)`, v will be 5.

Also, we can stack this. If we have an Option[Option[Int]] we can stack patterns.

```
val optionOption5: Option[Option[Int]]

optionOption5 match {
    case Some(Some(5)) => Do something when both options are some and the value is
    case Some(Some(a)) => Do something when both options are some and the value is
    case Some(None) => Do something when the inner option is not present
    case None => Do something when the outer option is not present
}
```

Exercise

1. create a function `shout` that retrieves a Pet as an argument and shouts the appropriate value to the console. NOTE: this usually breaks referential transparancy, but it's oke for demo purposes. The function `println` prints to the console.

Recursion

In this chapter, we are going to use are gained knowledge of ADT and pattern matching and use recursion, which is the way of iterating over a sequence. Although recursion is supported in other paradigms, it's not often used. One of the reason is the absence of tail-call elimination, which will be explained later as it's crucial for functional languages.

A recursive function is a function that has a reference call to itself. We can use an ADT again to create a linked list

```scala
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[A](value: A, tail: List[A]) extends List[A]
```

This is an recursive ADT. It can be read as following: A list of type A is either Nil (which represents an empty list) or a Cons with a value of type A and a tail (which is a list of type A itself).

With this ADT we are able to represent a list

```scala
Cons(1, Cons(2, Cons(3, Nil))) // A list with 3 elements: 1, 2, 3
```

We can write a recursive function to calculate the length of a list. As you can see in the implementation of length, it calls itself in the body of the function.

```scala
def length[A](list: List[A]): Int = list match {
  case Nil => 0
  case Cons(_, tail) => 1 + length(tail) // The recursive call to length
}
```

This function take a list as input. Whenever the pattern corresponds with Cons, we will return 1 + length(tail). If the value corresponds with Nil it will return a 0.

In functional programming, we can use the substitution model where we evaluate the expressions to values. We can do this because no functions have side-effects; same input returns to same output and can be substituted by it's expected result.

We can use the substitution model to evaluate this expression.

```
Expression:
length(Cons(1, Cons(2, Cons(3, Nil))))
```

```
1. length(Cons(1, Cons(2, Cons(3, Nil))))
2. 1 + length(Cons(2, Cons(3, Nil))))
3. 1 + (1 + length(Cons(3, Nil)))
4. 1 + (1 + (1 + 0))
5. 3
```

Recursive functions have some laws that they must obey:

- A recursive algorithm must have a base case.
- A recursive algorithm must change its state and move toward the base case.
- A recursive algorithm must call itself, recursively.

The base case is the condition that stops the recursion. In the case of the length function, the base case is evaluating to 0 whenever the list is empty. This will stop the recursion.

The second law states that the state must change and move towards the base case. Looking at the length function, the tail is used inside the recursive function instead of the entire list. The list becomes smaller and gets closer to the empty list.

The last law is obvious, which means that a recursive function must call itself.

Practice, practice and practice even more!

Exercises

- Write a function that takes a List[Int] and returns the sum value. Use the following signature:

```
def sum(list: List[Int]): Int
```

- Write a function called map which takes a list and a HOF that will transform every element of the list by applying the function:

```
def map[A, B](list: List[A])(f: A => B): List[B]
```

- Write a function called filter that takes a list and a predicate and returns a list with all elements where the predicate holds:

```
// This is the scala boolean, not the earlier used boolean
def filter[A](list: List[A])(f: A => Boolean): List[A]
```

- Write a function called any which verifies that atleast one element holds to the predicate function:

```
def any[A](list: List[A])(f: A => Boolean): Boolean
```

- Write a function called all which verifies that all elements hold to the predicate function:

```
def all[A](list: List[A])(f: A => Boolean): Boolean
```

- Write a function called concat that concatenates 2 lists:

```
def concat[A](l1: List[A], l2: List[A]): List[A]
```

- Write a function called flatMap that takes a list and a HOF of type `A => List[B]` and returns a List[B]. It concatenates both lists. You are allowed to use the concat function written before:

```
def flatMap[A, B](list: List[A])(f: A => List[B]): List[B]
```

tail-recursion

Tail-recursion is a special form of recursion, which is very important in functional programming. To demonstrate it's important, go run the sum function written before with the following argument: (1 until 10000 toList). Before you continue, what happened?

We got a StackOverflow exception. Does this mean we can't do these kinds of things in functional programming? Ofcourse not, tail-recursion to the rescue!

So what exactly happened when we ran the function with a list from 1 to 10000? Every time you call a function, a stack frame will be put onto the stack. A stack is a data structure where you can put data on, but only the item put on the latest can be taken off it. So every time a function is called, a stack frame is added onto the stack containing information about the function call. In our case, we got a recursive function, which will generate a new stackframe work for every recursive call and this will eventually exceed the limit.

Tail-recursion is a recursive function where the recursive call is the absolute latest thing the function does. Take another look at the sum example:

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

In this case the function can only complete when in completes the recursive call and afterwards still execute the addition to it's current item. If we use the substitution model we get the following result

```
1. sum(Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Nil)))))))
2. 1 + sum(Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Nil)))))))
3. 1 + (2 + sum(Cons(3, Cons(4, Cons(5, Cons(6, Nil))))))
4. 1 + (2 + (3 + sum(Cons(4, Cons(5, Cons(6, Nil))))))
5. 1 + (2 + (3 + (4 + sum(Cons(5, Cons(6, Nil))))))
6. 1 + (2 + (3 + (4 + (5 + sum(Cons(6, Nil))))))
7. 1 + (2 + (3 + (4 + (5 + (6 + sum(Nil))))))
8. 1 + (2 + (3 + (4 + (5 + (6 + 0)))))
9. 21
```

The expression is constantly expanding, where at the end when eventually all the recursive calls are executed, the expression needs to be evaluated as a whole, in the previous example this is step 8 to 9.

We can implement sum in tail form as following

```
def sum(list: List[Int]): Int = {
  def loop(list: List[Int], state: Int) = list match {
    def Nil => state
    def x :: xs => loop(xs, state + x)
  }

  loop(list, 0)
}
```

Let's do the same as we did with the previous example. We can use the substitution model to evaluate all the expressions.

```
1. sum(Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Nil)))))))
2. loop(Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Nil)))))), 0)
3. loop(Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Nil))))), 1)
4. loop(Cons(3, Cons(4, Cons(5, Cons(6, Nil)))), 3)
5. loop(Cons(4, Cons(5, Cons(6, Nil))), 6)
6. loop(Cons(5, Cons(6, Nil)), 10)
7. loop(Cons(6, Nil)), 15)
8. loop(Nil, 21)
9. 21
```

As you can see, when we apply the substitution model on a tail recursive function, the function is rewritten instead of expanded every recursive call. We accumulate the state in the recursive function. We can say that in terms of loop, loop(Cons(6, Nil)), 15) === loop(Nil, 21). These are steps 7 and 8 from the evaluation of the substitution model.

Hands-on

Abstracting over the pattern (head- and tail recursion)

Before you continue reading, do you see how we can abstract over these kinds of functions? Hint 1: What is the similarity between the product and sum functions written earlier?

Let's take another look at the implementations:

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

def product(list: List[Int]): Int = list match {
  case Nil => 1
  case x :: xs => x + product(xs)
}
```

We can see that the pattern is the same, only 2 things are different.

- The value when the list is empty
- The function that combines the head of the list to the result of the recursive part.

To abstract over this pattern, we can paramaterize the things that can change, which will result in the following signature. We call this function foldLeft

```
def foldLeft[A](list: List[A])(z: A)(f: (A, A) => A): A
```

We can even make this function more generic, since we can remove the fact that the output type needs to be the same type as the input type. The new signature will look like this

```
def foldLeft[A, B](list: List[A])(z: B)(f: (B, A) => B): B
```

Try to implement this function. As you are implementing these kinds of abstractions, you will notice that there are not that many ways to implement this. As functional programmers say, just follow the types!

We just implemented a function that can reduce a list of values to a single value. We can also implement a function which folds from the right side. This can be very important, since there are functions where order matters. Take subtraction for example with List(1, 2, 3). Folding from the left with the first element (1) as Z will result in (-4) where folding from the right with 3 as Z will result in 0.

Try implement foldRight now. The signature is as following:

```
def foldRight[A, B](list: List[A])(z: B)(f: (A, B) => B): B
```

Try to give a smart solution, so don't reverse the list at first!

We are not done yet, after we have discovered typeclasses, we can go a step further and abstract to Foldables!

Typeclass

Typeclasses are simply said better interfaces. Interfaces fully work based on the subtyping system that object-oriënted languages offer. I can implement the interface on classes which then needs to implement the methods on the interface. There is a simple problem with interfaces, which is that we can't implement inferfaces on classes that we didn't create. For example, if we use a library where we want to have a value object that implements a certain interface. This is simply not possible. Hence, typeclasses solve this problem and more.

In functional programming, we don't use this sub-typing. Simply said, data and things that can be done with the data are two seperate things. As an example we are going to take a look at a very simple example.

```
trait Eq[A] {
  def equal(x: A, y: A): Bool
}
```

We can implement this trait for every type A. For example, we can implement it for a String, Int and an Option[Int] type

```
val stringEq = new Eq[String] {
  def equal(x: String, y: String): Bool = x.equals(y)
}

val intEq = new Eq[Int] {
  def equal(x: Int, y: Int): Bool = x == y
}

val optionIntEq = new Eq[Option[Int]] {
  def equal(x: Option[Int], y: Option[Int]): Bool = (x, y) match {
    case (Some(a), Some(b)) => implicitly[Eq[Int]].equal(a, b) //ignore this implic
    case (None, None) => true
    case _ => false
  }
}
```

Now that we have type class instances available, we are able to test equality of our different types that implement the type class. To test equality, you can just call the function

```
  intEq.equal(3, 4) // false
```

This is very nice, but it's very likely that we want to abstract upon typeclasses. How do we get our type class instances?

Implicits to the rescue! We can tell Scala that we want to implicitly rely that it exist. This will be verified at compile-time, so no runtime problems at all. We call this principle implicit evidence

```
trait Empty[A] {
    def empty: A
}

def helloWorldIfEmpty(a: A)(implicit eq: Eq[A], empty: Empty[A]): String = {
  if (eq.equal(a, empty.empty)) "Hello World!"
  else "......"
}
```

The implicit keyword can only be in the last parameter list. If you rely on multiple implicits, you can add them in the last argument list as shown in the example. In this case, the caller of the function don't need to provide how the value can be verified for equality and what the empty value is for a given type.

/* Explain implicit scope /*

We need to change 1 more thing to make this code compile. We need to make the instances implicit. You can do this by adding implicit in front of the val or def.

```
implicit val stringEq = new Eq[String]
```

Let's do some examples

```
implicit val intEq = new Eq[Int] {
  def equal(x: Int, y: Int): Bool = x == y
}

implicit val intEmpty = new Empty[Int] {
  def empty = 0
}

helloWorldIfEmpty(0) // "Hello World"
helloWorldIfEmpty(1) // "......"

-------------------------------------

implicit val stringEq = new Eq[String] {
  def equal(x: String, y: String): Bool = x.equals(y)
}

implicit val stringEmpty = new Empty[String] {
  def empty = ""
}

helloWorldIfEmpty("yadayada") // "......"
helloWorldIfEmpty("") // "Hello World"
```

What I personally really like about typeclasses is that they are existential. You can add functionality to your types which will really make sense if you use them more often!

Abstracting the length method using typeclasses

Remember that we wrote the length function when practicing our recursion skills. It looks like this:

```
def length[A](list: List[A]): Int = list match {
    case Nil => 0
    case x :: xs => 1 + length(xs)
}
```

We did write this function with the help of tail-recursion as well to make sure it's also able to compute the length of a really long list.

```
def length[A](list: List[A]): Int = {
    def loop(list: List[A])(acc: Int): Int = list match {
        case Nil => acc
        case x :: xs => loop(xs)(acc + 1)
    }
}
```

While we practiced our recursion skills, we discovered a pattern and factored this out and called it folding, or more specific foldLeft and foldRight. The length function with foldLeft did look like this:

```
def length[A](list: List[A]): Int = foldLeft(list)(0)((b, _) => b + 1)
```

But, we can go even further! Remember the tree? Isn't it weird that we are not able to compute the length of a tree? To do so, I need to write a new function.. Bummer! I wish there was something called.... typeclasses! There is a special typeclass that we can use perfectly suited for our needs. Take another look at our length function written with foldLeft. The problem is the hardcoded type-constructor that we can fold. The foldable typeclass looks like this:

```
trait Foldable[F[_]] {
    def foldLeft[A, B](fa: F[A])(z: B)(f: (B, A) => B): B
}
```

Exercise

1. Try to implement this Foldable typeclass for List and Binary Tree. Assume that they are endoded like this:

```
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[A](v: A, tail: => List[A]) extends List[A]

sealed trait Tree[+A]
case class Node[A](left: Tree[A], right: Tree[A]) extends Tree[A]
case class Leaf[A](v: A) extends Tree[A]
```

1. Try to make the length function work for both the List and Tree

```
def length[F[_], A](fa: F[A])(implicit foldable: Foldable[F]): Int =
    foldable.foldLeft(fa)(0)((b, _) => b + 1)
```