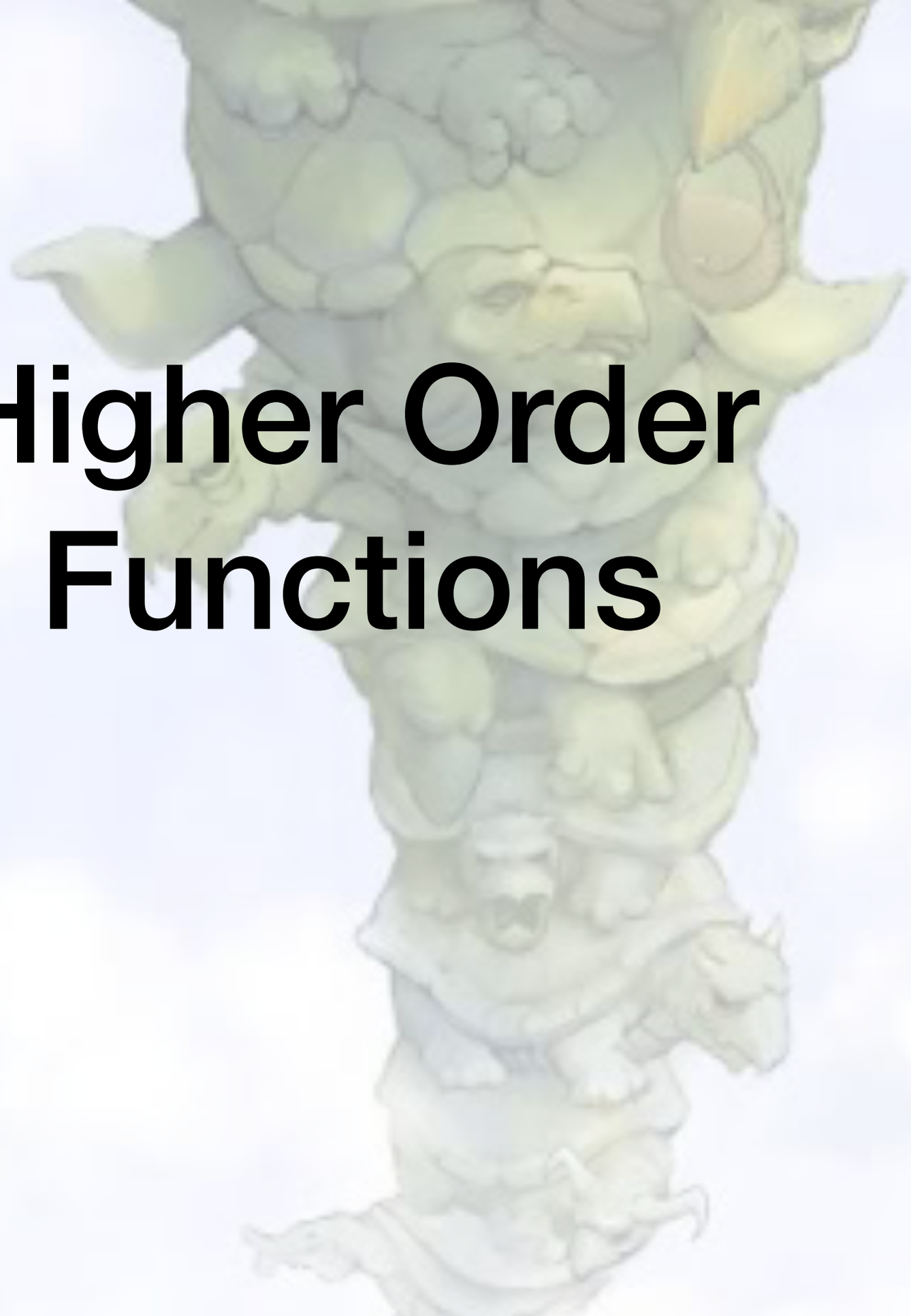
Higher Order Functions



First class citizen

- In Scala, a function is a first class citizen
- A function has a type, and is a value
- A function can be passed as an argument to other functions
- Functions that take other functions as arguments, or return functions as results are called **Higher Order Functions**

```
scala>
scala> def hello( stranger: String) = {"Hello, "+stranger}
hello: (stranger: String)String

scala> :t hello _
String => String
```

Example HOF

- When we are to lazy, to give names
- Syntactic sugar: ()=> instead of def
- (x:Int) function parameters
- x*x*x - function body

```
scala> (x:Int)=> x*x*x
res3: Int => Int = $$Lambda$1252/1407721609@66a8751a

scala> def f(x:Int)= x*x*x
f: (x: Int)Int
```

Problem [higher order]

- 1+2+3+4
- 1^2+ 2^2+3^2+4^2
- 1! + 2! + 3! + 4!
- What is the common pattern?
- How would you write this in Scala?

$$\sum_{k=a}^{b} f(k)$$

Can we reuse the common pattern?

```
scala> def sum(a:Int, b:Int):Int = if(a>b) 0 else a + sum(a+1,b)
sum: (a: Int, b: Int)Int

scala> sum(1,4)
res11: Int = 10
```

```
scala> def sumsq(a:Int, b:Int):Int = if(a>b) 0 else a*a +
sumsq(a+1,b)
sumsq: (a: Int, b: Int)Int

scala> sumsq(1,4)
res12: Int = 30res8: Int = 14
```

Solution - use a higher order function

```
scala> def sum(f: Int=>Int, a:Int, b:Int):Int = if(a>b) 0 else f(a)
+ sum(f, a+1,b)
sum: (f: Int => Int, a: Int, b: Int)Int

scala> sum( (a) => a, 1, 4)
res13: Int = 10

scala> sum( (a) => a*a, 1, 4)
res14: Int = 30
```

Function type

- A =>B is the type of a function, that takes an argument of type A, and returns a result of type B
- Fx, Int => Int is a function that maps integers to integers

Function literal

- Scala has first-class functions
- You define functions and call them
- you can write down functions as unnamed literals and then pass them around as values
- A function literal is compiled into a class that when instantiated at runtime is a function value.[
  - The distinction between function literals and values is that function literals exist in the source code, whereas function values exist as objects at runtime.
  - The distinction is much like that between classes (source code) and objects (runtime).

```
scala> var increase = (x: Int) => x + 1
  increase: Int => Int = <function1>

  scala> increase(10)
  res0: Int = 11
```

Shortcuts

- leave off the parameter types - can be inferred from context
- leave out parentheses around a parameter whose type is inferred
- use placeholder syntax - underscore as a "blank" in the expression that needs to be "filled in”

```
scala> someNumbers.filter(x => x > 0)
 res6: List[Int] = List(5, 10)
// placeholder syntax
someNumbers.filter(_ > 0)
```

Currying

In mathematics, **currying** is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.

$$h = (x \mapsto (y \mapsto f(x, y)))$$

Currying in Scala

In mathematics, **currying** is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.

Methods may define multiple parameter lists.

When a method is called with a fewer number of parameter lists, then this will yield a function taking the missing parameter lists as its arguments.

```
def filter(xs: List[Int], p: Int => Boolean): List[Int] =
    if (xs.isEmpty) xs
    else if (p(xs.head)) xs.head :: filter(xs.tail, p)
    else filter(xs.tail, p)

  def modN(n: Int)(x: Int) = ((x % n) == 0)

  val nums = List(1, 2, 3, 4, 5, 6, 7, 8)
  println(filter(nums, modN(2)))
  println(filter(nums, modN(3)))
```

Problem [currying]

- Credit card company has list of credit cards
- Calculate the premiums for all those cards that the credit card company has to pay out.
- Premiums depend on the total number of credit cards

```
case class CreditCard(creditInfo: CreditCardInfo, issuer: Person,
account: Account)

object CreditCard {
    def getPremium(totalCards: Int, creditCard: CreditCard): Double
= { ... }
}

//how to sum all the premiums?
//val creditCards: List[CreditCard] = getCreditCards()
//val allPremiums = creditCards.map(CreditCard.getPremium).sum
```

Credit: https://lukajcb.github.io/blog/scala/2016/03/08/a-real-world-currying-example.html

Use currying!

- Credit card
- premiums for credit card usage. What we have is a list of credit cards and we'd like to calculate the premiums for all those cards that the credit card company has to pay out. The premiums themselves depend on the total number of credit cards, so that the company adjust them accordingly.
- https://lukajcb.github.io/blog/scala/2016/03/08/a-real-world-currying-example.html

```
object CreditCard {
    def getPremium(totalCards: Int)(creditCard: CreditCard): Double
= { ... }
}

val creditCards: List[CreditCard] = getCreditCards()

val getPremiumWithTotal = CreditCard.getPremium(creditCards.length)_

val allPremiums = creditCards.map(getPremiumWithTotal).sum
```

Multiparameterlist

- syntax sugar
- sum(f) is a valid expression

```
scala> def sum(f: Int=>Int, a:Int, b:Int):Int = if(a>b) 0 else f(a)
+ sum(f, a+1,b)
sum: (f: Int => Int, a: Int, b: Int)Int

scala> :t sum _
(Int => Int, Int, Int) => Int

scala> def sum(f: Int=>Int)(a:Int, b:Int):Int = if(a>b) 0 else f(a)
+ sum(f)( a+1,b)
sum: (f: Int => Int)(a: Int, b: Int)Int

scala> :t sum _
(Int => Int) => ((Int, Int) => Int)
```

Lab Time!

higher_order_functions_04