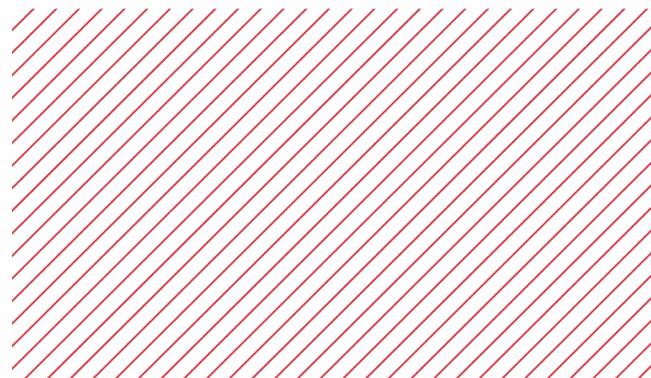


академия
больших
данных



Advanced Scala

Корольков Алексей



Структура курса

1. Введение в Scala. Основные конструкции языка



2. ООП в Scala. Pattern matching. Функциональные конструкции. Adt

3. Библиотека коллекций в Scala



4. Асинхронные операции, обработка исключений, неявные параметры

5. Параметрический полиморфизм. Имплиситы



6. Основы функционального программирования. Часть 1

7. Основы функционального программирования. Часть 2

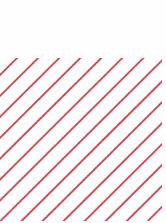
8. ZIO, TF и основной ФП-стэк в Scala

9. Функциональные стримы на примере fs2, работа с бд

10. Акторы в Scala



Implicits



Implicit parameters

```
def map[S](f: T => S)(implicit executor: ExecutionContext): Future[S]
```

По каким критериям компилятор ищет подходящий *implicit*?

- Именованный объект, помеченный ключевым словом *implicit*
- Тип этого объекта должен соответствовать типу **T**
- Этот объект должен быть в **implicit** скоупе



Implicit scope

- Локальный скоп по месту вызова
- Компаньон объект для исходного типа
- Импорт



Implicit conversions

Дан **class** C и **trait** T, как сделать так, чтобы C мог экстендер T, при этом изменять код C нельзя?

```
trait T  
class C  
implicit def c2t(c: C): T = new T{}
```



High-kinded types

Higher-Kinded type - тип который абстрагируется над конструктором типа

```
trait M[F[_]]
```



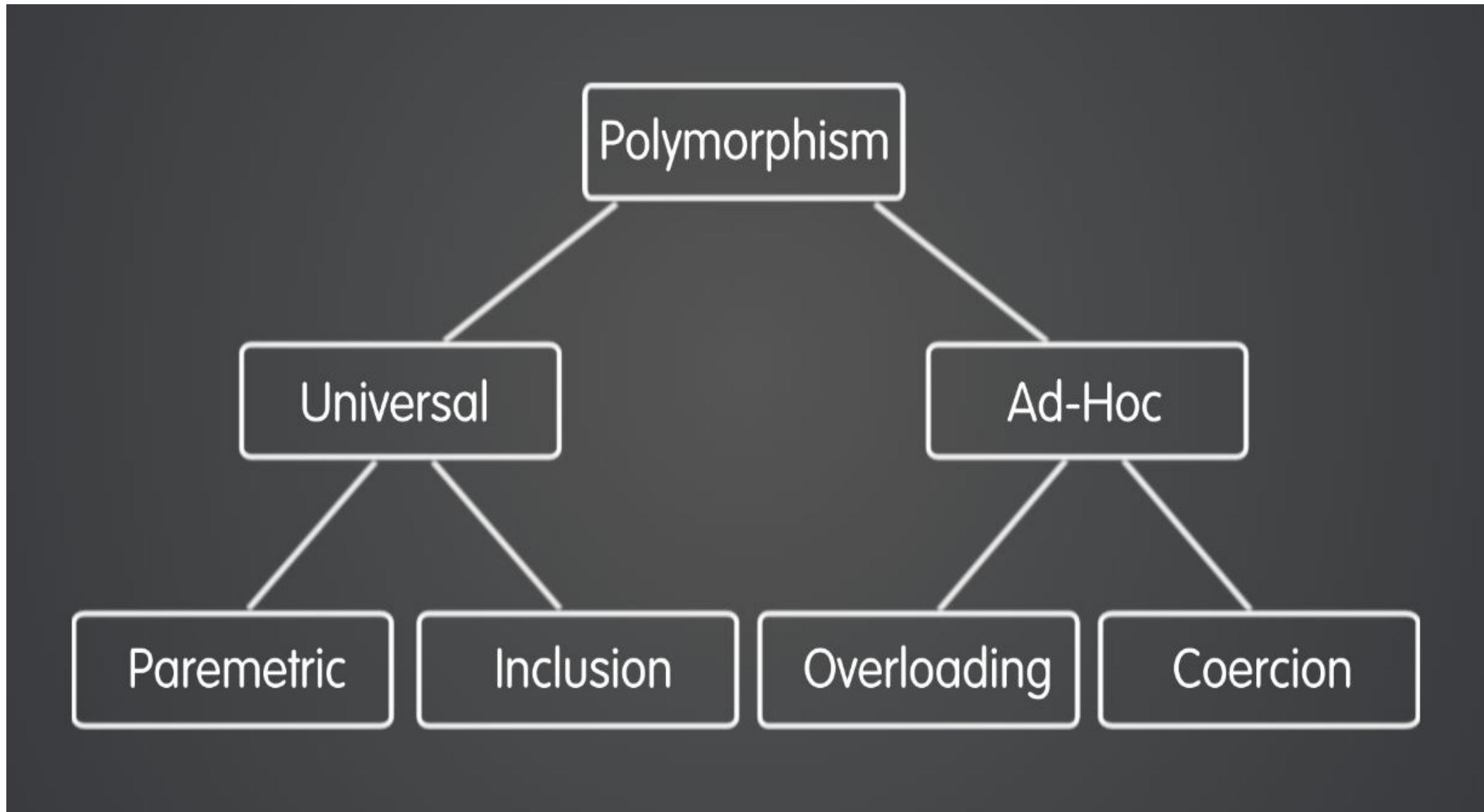
Polymorphism

Polymorphism

Полиморфизм - это особенность языка программирования, которая позволяет один интерфейс использовать для общего класса действий. Это понятие часто выражается как «один интерфейс, несколько действий».

[Theories of Programming Languages: Reynolds, John C](#)

Classifications of polymorphism



Inclusion

Включение - это форма полиморфизма, в которой подтип является типом данных, который связан с другим типом данных (супертиповом) некоторым понятием заменяемости.

Subtype polymorphism:

```
trait Animal
trait Furry extends Animal
trait HasLegs extends Animal
trait FourLegged extends Animal with HasLegs
class Cat extends Animal with Furry with FourLegged

val cat = new Cat()
```

Overloading

Перегрузка - позволяет создавать несколько методов с одинаковыми сигнатурами, которые отличаются друг от друга типом входных и выходных аргументов.

Operators overload:

```
case class Re(value: Double) {  
    def +(another: Re): Re = copy(value + another.value)  
}
```

```
case class Im(value: Double) {  
    def +(another: Im): Im = copy(value + another.value)  
}
```

```
case class Complex(re: Re, im: Im) {  
    def +(another: Complex): Complex =  
        Complex(re + another.re, im + another.im)
```

```
    def +(anotherRe: Re): Complex =  
        copy(re = re.copy(re.value + anotherRe.value))
```

```
    def +(anotherIm: Im): Complex =  
        copy(im = im.copy(im.value + anotherIm.value))  
}
```

Coercion

Coercion - это операция преобразования аргумента или операнда в тип, ожидаемый функцией или оператором.

Coercion by implicit conversions:

```
case class Complex(re : Double, im : Double) {  
    def + (another : Complex): Complex =  
        Complex(re + another.re, im + another.im)  
}
```

//implicit conversion: a form of coercion
implicit def doubleToComplex(d: Double): Complex = Complex(d, 0)

```
Complex(2.0, 5.0) + 5.0 // == Complex(7.0, 5.0)  
5.0 + Complex(1.0, -2.0) // == Complex(6.0, -2.0)
```

Ad-hoc. Type-classes

Классы типов - это мощная и гибкая концепция, которая добавляет в Scala ad-hoc полиморфизм. Они не встроены в язык, в Scala их эмулируют с помощью неявных параметров.

In Haskell:

```
class Show a where  
    show :: a -> String
```

In Scala:

```
trait Show[A] {  
    def shows(a: A): String  
}
```

//OR Haskell notation:

```
trait Show[A] {  
    def shows : A => String  
}
```

Ad-hoc. Type-classes

Type-classes implementation:

```
// Type-class Int instance
implicit val IntShow = new Show[Int] {
  def shows(a : Int) = a.toString
}

//Type-class List instance:
implicit def ListShow[T] = new Show[List[T]] {
  def shows(a : List[T]) = a.mkString(", ")
```

Using a Type-classes:

```
def shows[A](a : A)(implicit sh: Show[A]) = sh.shows(a)
//OR:
def shows[A : Show](a : A) = implicitly[Show[A]].shows(a)
```

// must have a "Show[Int]" instance in scope
shows(42)

Parametric (Type as parameter)

Абстракция над параметром типа позволяет писать функции работающие для произвольных типов

```
class Container[A](val value : A) {           // Type-constructor
  def map[B](f : A => B): Container[B] =
    new Container(f(value))
}
```

```
new Container(10) map { x => (x * 2).toString } // Container("20")
new Container("10") map { x => x.toInt * 2 }   // Container(20)
```

Higher-kinded (Ranked)

Kind is the type of a type-constructor (type of a higher-order type operator)

Value level:

```
val x : String  
val y : List[String]
```

```
def id(x : Int):Int = x  
def map(f : Int=> Int, x: Int) = f(x)
```

Type level:

```
type String    :: *  
type List      :: * -> *  
type Function1 :: * -> * -> *
```

```
type Id[A] = A  
type Map[C[_],A] = C[A]
```

Спасибо за внимание