

Implicits and Typeclasses





Хакимов Айнур

- Scala разработчик в Tinkoff
- До Tinkoff писал на C/C++

План

1. Вводная

2. Базовый синтаксис

3. Функции

4. ADT, Collections

5. Implicits,
Typeclasses

6. Common Typeclasses

7. Monad

8. Асинхронное
выполнение

9. Future

10. Монада IO

План лекции

План лекции

- Implicit conversions

План лекции

- Implicit conversions
- Implicit parameters

План лекции

- Implicit conversions
- Implicit parameters
- Implicit classes

План лекции

- Implicit conversions
- Implicit parameters
- Implicit classes
- Type classes

План лекции

- Implicit conversions
- Implicit parameters
- Implicit classes
- Type classes
- Simple type classes

Implicit conversions

```
val x: String = 123
```

Скомпилируется ли такой код?

```
val x: String = 123

// [error] ....: type mismatch;
// [error] found   : Int(123)
// [error] required: String
// [error]   val x: String = 123
// [error]                           ^
// [error] one error found
// [error] (Compile / compileIncremental) Compilation failed
```

Implicit conversions

```
1 implicit def intToString(x: Int): String = x.toString
2
3 val x: String = 123 // будет вызвана неявная функция intToString
4
5 // x: "123"
```

Implicits

```
implicit def intToString1(x: Int): String = x.toString
implicit def intToString2(x: Int): String = x.toString

val x: String = 123

// [error] Note that implicit conversions are not applicable because they are ambiguous:
// [error] both method intToString1 in object Example of type (x: Int): String
// [error] and method intToString2 in object Example of type (x: Int): String
// [error] are possible conversion functions from Int(123) to String
```

Implicits

Название неявной функции нужно в двух ситуациях

1. Если нужно вызвать функцию явно

```
1 implicit def intToString(x: Int): String = x.toString  
2 val x: String = intToString(123)
```

Implicits

Название неявной функции нужно в двух ситуациях

2. Для явного импорта

```
1 object my_implicitlys {  
2     implicit def intToString(x: Int): String = x.toString  
3 }  
4 import my_implicitlys.intToString  
5 val x: String = 123
```

Implicits

Название неявной функции нужно в двух ситуациях

2. Для явного импорта

```
1 object my_implicitlys {  
2     implicit def intToString(x: Int): String = x.toString  
3 }  
4 import my_implicitlys.intToString  
5 val x: String = 123
```

Implicits

Для определения неявных функций:

- Нужно использовать ключевое слово `implicit`
- Это функция должна быть объявлена в `trait`/`class`/`object`/`method`
- Должен быть только один параметр в списке аргументов

Implicits

```
implicit def func(paramA: A, paramB: B): C = ???
```

```
// такая функция не будет вызываться неявно
```

Implicit scopes and priorities

Implicit scopes and priorities

Local scope

```
1 object Example {  
2  
3     implicit def intToString(x: Int): String = x.toString  
4  
5     val x: String = 123  
6 }
```

Implicit scopes and priorities

Imports

```
1 object ExternalImplicits {  
2     implicit def intToString(x: Int): String = x.toString  
3 }  
4  
5 object Example {  
6  
7     import ExternalImplicits.intToString  
8     // OR  
9     import ExternalImplicits._  
10  
11    val x: String = 123  
12 }
```

Implicit scopes and priorities

Объекты-компаньоны

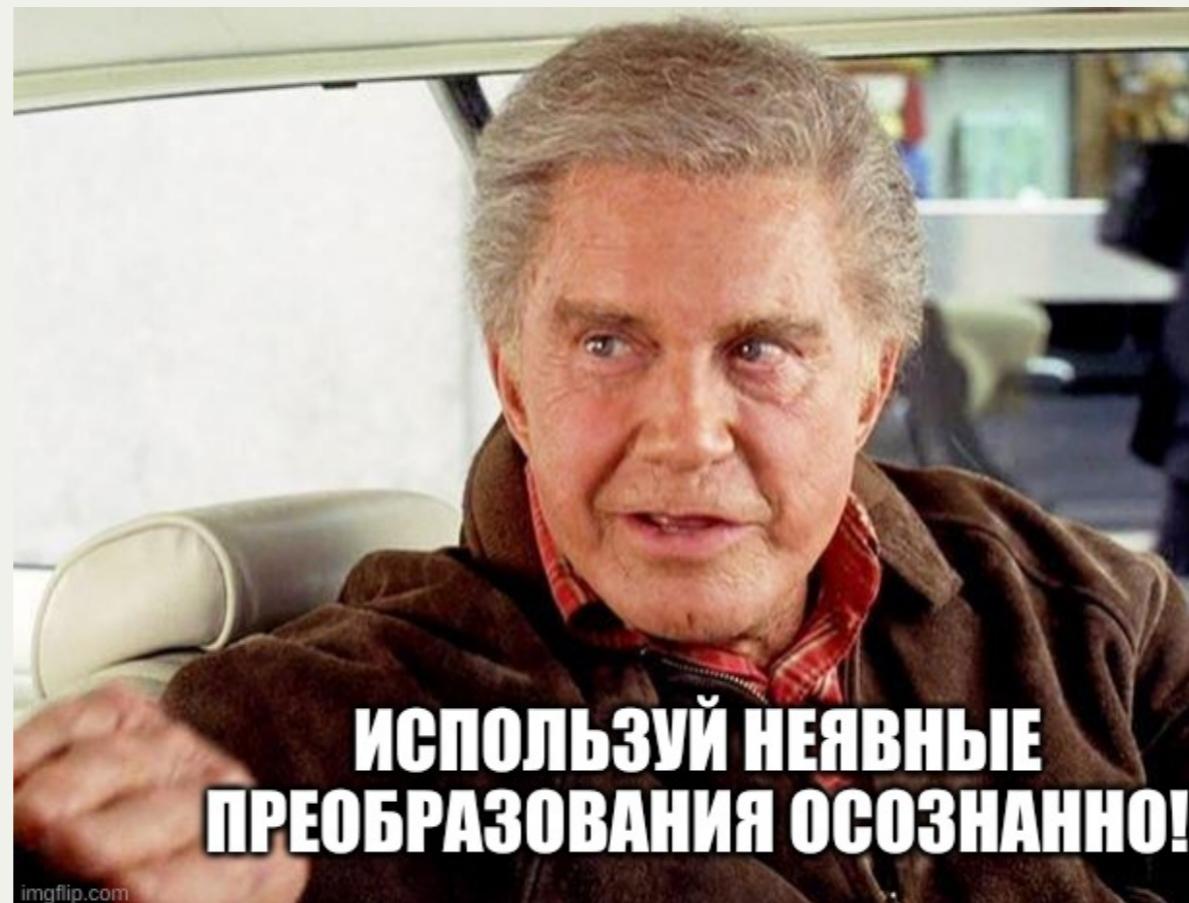
```
1 trait Currency
2 case class Dollar(amount: Double) extends Currency
3 case class Euro(amount: Double) extends Currency
4
5 object Currency {
6     implicit def euroToDollar(euro: Euro): Dollar = Dollar(euro.amount * 1.13)
7 }
8
9 object Example extends App {
10
11     val dollar: Dollar = Euro(100) // euroToDollar
12 }
```

Implicit scopes and priorities

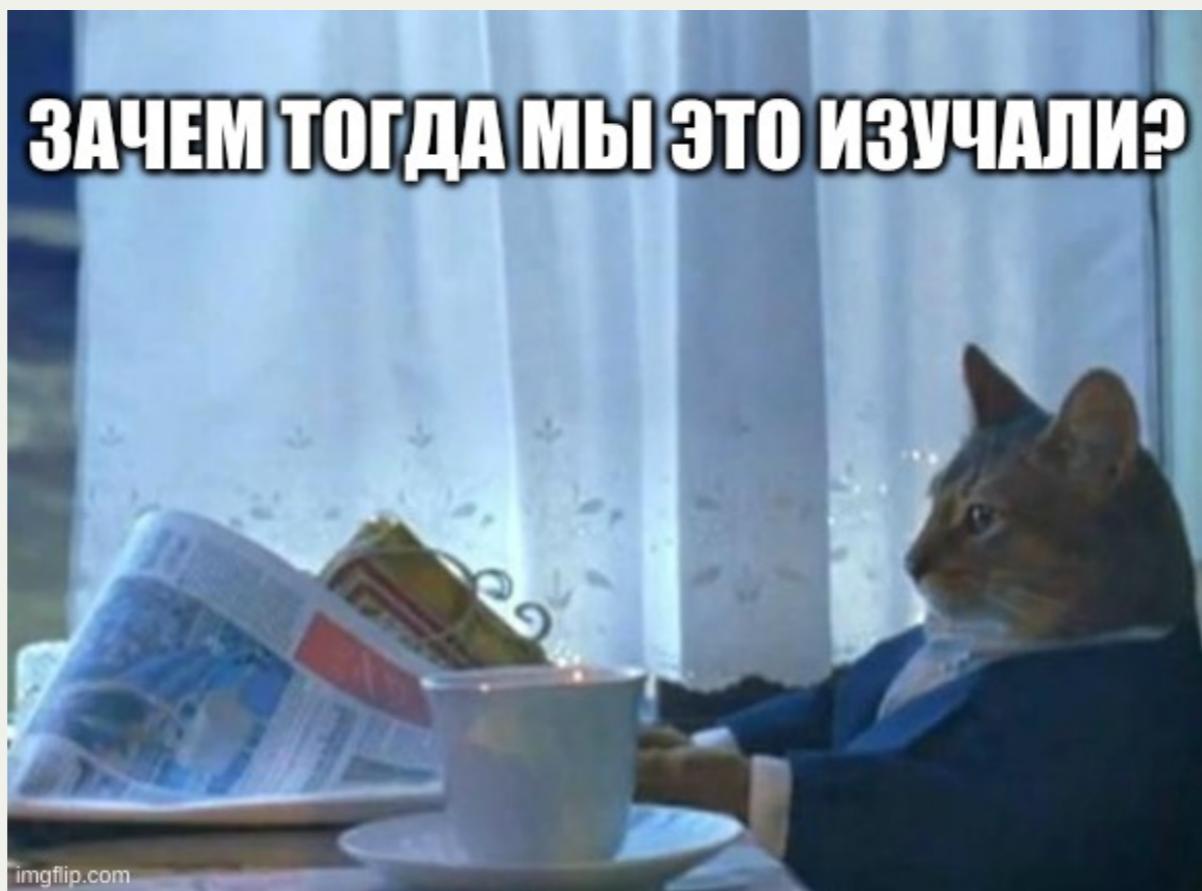
Объекты-компаньоны

```
1 trait Currency
2 case class Dollar(amount: Double) extends Currency
3 case class Euro(amount: Double) extends Currency
4
5 object Currency {
6     implicit def euroToDollar(euro: Euro): Dollar = Dollar(euro.amount * 1.13)
7 }
8
9 object Example extends App {
10
11     implicit def euroToDollar(euro: Euro): Dollar = Dollar(euro.amount)
12
13     val dollar: Dollar = Euro(100) // ???
14 }
```

Предостережение



Неосознанное использование неявных преобразований может привести написанию трудного для понимания кода.



Вопросы?

Implicit parameters

```
def func(implicit x: Int): Unit = ???
```

Implicit parameters

```
1 def multiply(x: Int)(implicit y: Int) = x * y
2
3 implicit val z: Int = 10 // должна быть неявной
4
5 multiply(3) // result: 30
6 multiply(4) // result: 40
```

Implicit parameters

```
1 def multiply(x: Int)(implicit y: Int) = x * y
2
3 implicit val z: Int = 10 // должна быть неявной
4
5 multiply(3) // result: 30
6 multiply(4) // result: 40
```

Implicit parameters

```
1 def multiply(x: Int)(implicit y: Int) = x * y
2
3 implicit val z: Int = 10 // должна быть неявной
4
5 multiply(3) // result: 30
6 multiply(4) // result: 40
```

Implicit parameters

```
1 def multiply(x: Int)(implicit y: Int) = x * y
2
3 implicit val z: Int = 10 // должна быть неявной
4
5 multiply(3) // result: 30
6 multiply(4) // result: 40
```

Implicit parameters

```
implicit val z: Int = 10
implicit val y: Int = 42

multiply(3)

// [error] ....scala:119:11: ambiguous implicit values:
// [error] both value z in object ExampleImplicitParameters of type Int
// [error] and value y in object ExampleImplicitParameters of type Int
// [error] match expected type Int
// [error]   multiply(3)
```

Implicit parameters

```
1 def function(a: Int)(b: Int, c: Int)(implicit d: Int, e: Int) = ???
```

Implicit parameters

Примеры объявления

Implicit parameters

Примеры объявления

- `def func(implicit x: Int)` - x is implicit

Implicit parameters

Примеры объявления

- `def func(implicit x: Int)` - x is implicit
- `def func(implicit x: Int, y: Int)` - x and y are implicit

Implicit parameters

Примеры объявления

- `def func(implicit x: Int)` - x is implicit
- `def func(implicit x: Int, y: Int)` - x and y are implicit
- `def func(x: Int, implicit y: Int)` - wont compile!

Implicit parameters

Примеры объявления

- `def func(implicit x: Int)` - x is implicit
- `def func(implicit x: Int, y: Int)` - x and y are implicit
- `def func(x: Int, implicit y: Int)` - **wont compile!**
- `def func(x: Int)(implicit y: Int)` - only y is implicit

Implicit parameters

Примеры объявления

- `def func(implicit x: Int)` - x is implicit
- `def func(implicit x: Int, y: Int)` - x and y are implicit
- `def func(x: Int, implicit y: Int)` - **wont compile!**
- `def func(x: Int)(implicit y: Int)` - only y is implicit
- `def func(implicit x: Int)(y: Int)` - **wont compile!**

Implicit parameters

Примеры объявления

- def func(implicit x: Int) - x is implicit
- def func(implicit x: Int, y: Int) - x and y are implicit
- def func(x: Int, implicit y: Int) - **wont compile!**
- def func(x: Int)(implicit y: Int) - only y is implicit
- def func(implicit x: Int)(y: Int) - **wont compile!**
- def func(implicit x: Int)(implicit y: Int) - **wont compile!**

Implicit parameters

```
1 case class RequestContext(requestId: String)
2
3 class Logger {
4     def log(msg: String)(ctx: RequestContext): Unit = println(s"[` ${ctx.requestId} ] $`$msg")
5 }
6
7 object SomeApplication extends App {
8     val logger = new Logger()
9     def handle(requestContext: RequestContext) = {
10         logger.log("Starting process")(requestContext)
11         // some action ...
12         logger.log("Continue process...")(requestContext)
13         // some action ...
14         logger.log("End process")(requestContext)
15     }
16 }
```

Implicit parameters

```
1 case class RequestContext(requestId: String)
2
3 class Logger {
4     def log(msg: String)(ctx: RequestContext): Unit = println(s"[` ${ctx.requestId} ] $`$msg")
5 }
6
7 object SomeApplication extends App {
8     val logger = new Logger()
9     def handle(requestContext: RequestContext) = {
10         logger.log("Starting process")(requestContext)
11         // some action ...
12         logger.log("Continue process...")(requestContext)
13         // some action ...
14         logger.log("End process")(requestContext)
15     }
16 }
```

Implicit parameters

```
1 case class RequestContext(requestId: String)
2
3 class Logger {
4     def log(msg: String)(ctx: RequestContext): Unit = println(s"[` ${ctx.requestId} ] $`$msg")
5 }
6
7 object SomeApplication extends App {
8     val logger = new Logger()
9     def handle(requestContext: RequestContext) = {
10         logger.log("Starting process")(requestContext)
11         // some action ...
12         logger.log("Continue process...")(requestContext)
13         // some action ...
14         logger.log("End process")(requestContext)
15     }
16 }
```

Implicit parameters

```
2
3 class Logger {
4     def log(msg: String)(ctx: RequestContext): Unit = println(s"[` ${ctx.requestId}] `$msg")
5 }
6
7 object SomeApplication extends App {
8     val logger = new Logger()
9     def handle(requestContext: RequestContext) = {
10         logger.log("Starting process")(requestContext)
11         // some action ...
12         logger.log("Continue process...")(requestContext)
13         // some action ...
14         logger.log("End process")(requestContext)
15     }
16 }
```

Implicit parameters

```
1 case class RequestContext(requestId: String)
2
3 class Logger {
4     def log(msg: String)(implicit ctx: RequestContext): Unit = println(s"[` ${ctx.requestId} ] $`$msg")
5 }
6
7 object SomeApplication extends App {
8     val logger = new Logger()
9     def handle(implicit requestContext: RequestContext) = {
10         logger.log("Starting process")
11         // some action ...
12         logger.log("Continue process...")
13         // some action ...
14         logger.log("End process")
15     }
16 }
```

Implicit parameters

```
1 case class RequestContext(requestId: String)
2
3 class Logger {
4     def log(msg: String)(implicit ctx: RequestContext): Unit = println(s"[` ${ctx.requestId} ] $`$msg")
5 }
6
7 object SomeApplication extends App {
8     val logger = new Logger()
9     def handle(implicit requestContext: RequestContext) = {
10         logger.log("Starting process")
11         // some action ...
12         logger.log("Continue process...")
13         // some action ...
14         logger.log("End process")
15     }
16 }
```

Implicit parameters

```
1 case class RequestContext(requestId: String)
2
3 class Logger {
4     def log(msg: String)(implicit ctx: RequestContext): Unit = println(s"[` ${ctx.requestId} ] $`$msg")
5 }
6
7 object SomeApplication extends App {
8     val logger = new Logger()
9     def handle(implicit requestContext: RequestContext) = {
10         logger.log("Starting process")
11         // some action ...
12         logger.log("Continue process...")
13         // some action ...
14         logger.log("End process")
15     }
16 }
```

Implicit parameters

```
2
3 class Logger {
4     def log(msg: String)(implicit ctx: RequestContext): Unit = println(s"[` ${ctx.requestId}] $`msg")
5 }
6
7 object SomeApplication extends App {
8     val logger = new Logger()
9     def handle(implicit requestContext: RequestContext) = {
10         logger.log("Starting process")
11         // some action ...
12         logger.log("Continue process...")
13         // some action ...
14         logger.log("End process")
15     }
16 }
```

Вопросы?

Implicit classes

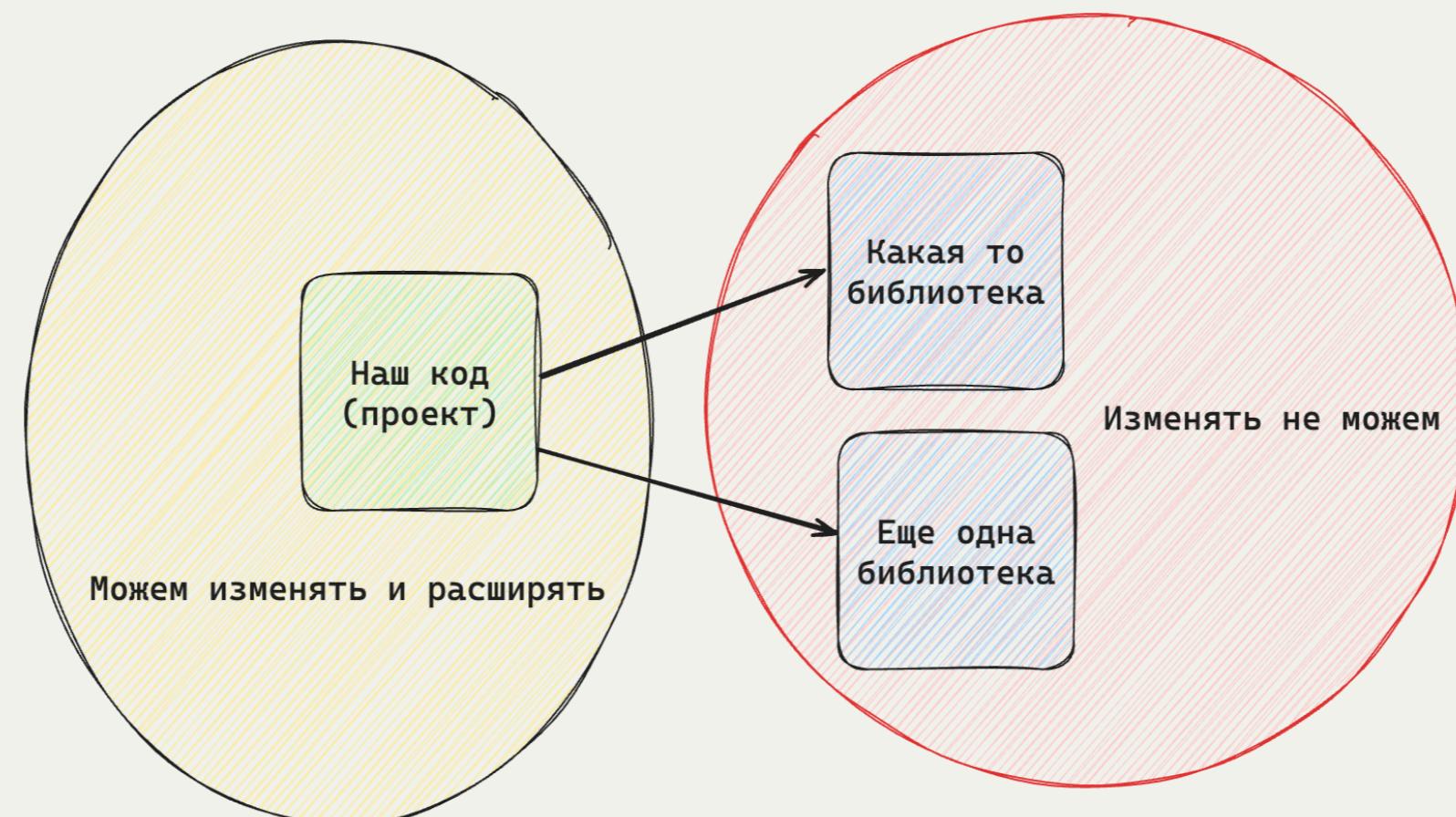
```
implicit class ImplicitClass(val field: Int) extends AnyVal {  
    def extention: Unit = ???  
}
```

Implicit classes

```
implicit class ImplicitClass(val field: Int) extends AnyVal {  
    def extention: Unit = ???  
}
```

Зачем они нужны?

Разница между нашим и не нашим кодом



ООП подход

Паттерн Adapter

```
1 class IntAdapter(val i: Int) {  
2     def isEven: Boolean = i % 2 == 0  
3     def isOdd: Boolean = !isEven  
4 }  
5  
6 // Создание экземпляра адаптера и использование его методов  
7 new IntAdapter(42).isEven // true  
8 new IntAdapter(42).isOdd // false
```

Implicit classes

```
1 implicit class RichInt(val i: Int) extends AnyVal {  
2   def isEven: Boolean = i % 2 == 0  
3   def isOdd: Boolean = !isEven  
4 }  
5  
6 // Можем вызывать методы  
7 42.isEven // true  
8 42.isOdd // false
```

А что там под капотом?



```
1 implicit class RichInt(val i: Int) extends AnyVal {  
2   def isEven: Boolean = i % 2 == 0  
3   def isOdd: Boolean = !isEven  
4 }  
5  
6 // Обессахаренный код  
7 Example.RichInt(42).isEven // true  
8 Example.RichInt(42).isOdd // false
```

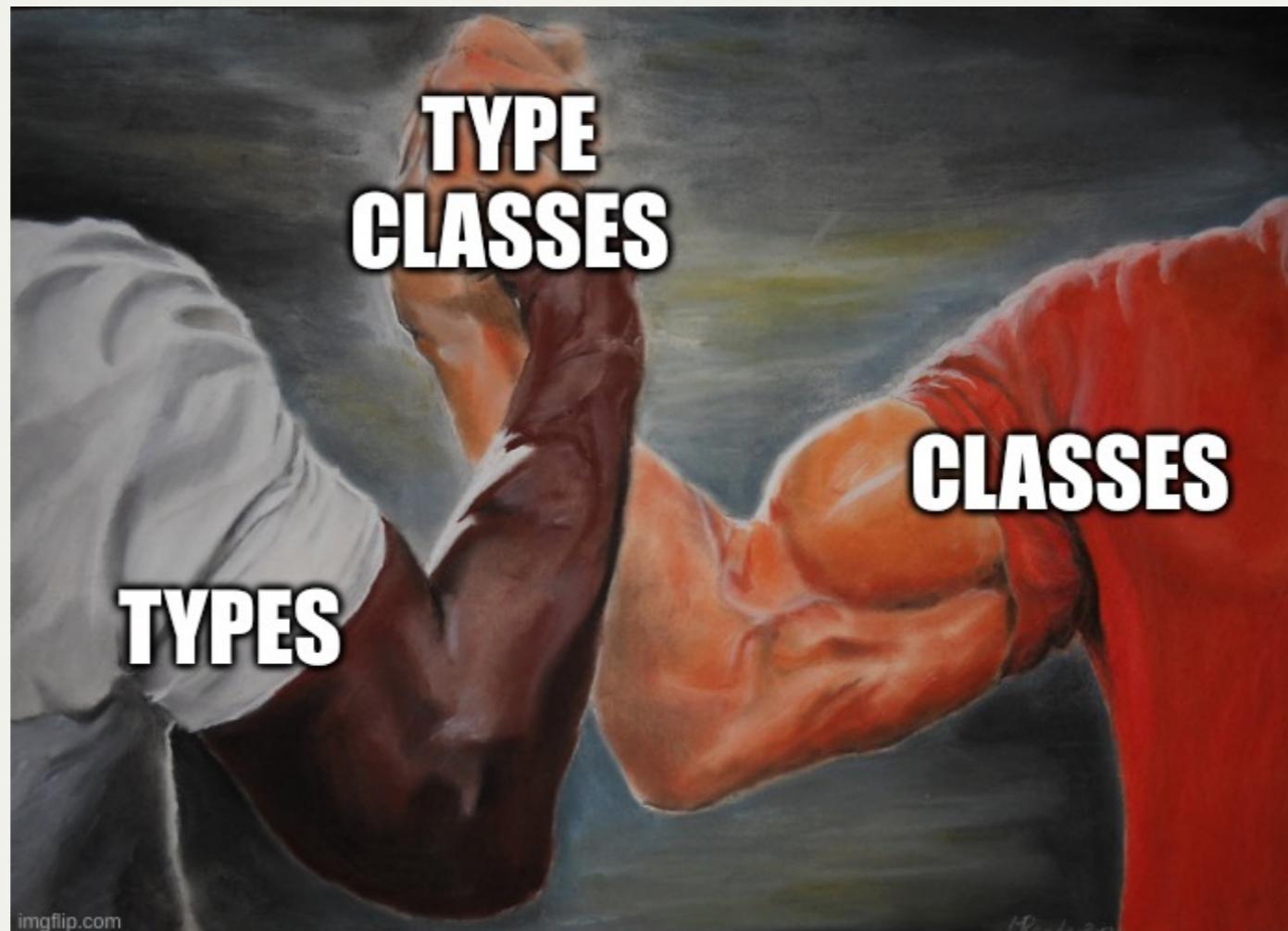
Имплиситные классы позволяют нам расширять функционал типов и классов не прибегая к наследованию или изменению

Implicit classes

- `extends AnyVal` в Scala используется для создания **value classes**, которые представляют собой механизм оптимизации, позволяющий избежать выделения памяти

Вопросы?

Type classes



Type classes

Тайпкласс - это паттерн, используемый в функциональном программировании для обеспечения Ad-hoc полиморфизма

Type classes

Тайпкласс - это паттерн, используемый в функциональном программировании для обеспечения Ad-hoc полиморфизма

Зачем они нужны? Причем тут полиморфизм?

Полиморфизм через наследование

```
1 trait Area {  
2     def area: Double  
3 }  
4  
5 class Circle(radius: Double) extends Area {  
6     override def area: Double = math.Pi * math.pow(radius, 2)  
7 }  
8  
9 class Rectangle(width: Double, length: Double) extends Area {  
10    override def area: Double = width * length  
11 }
```

Полиморфизм через наследование

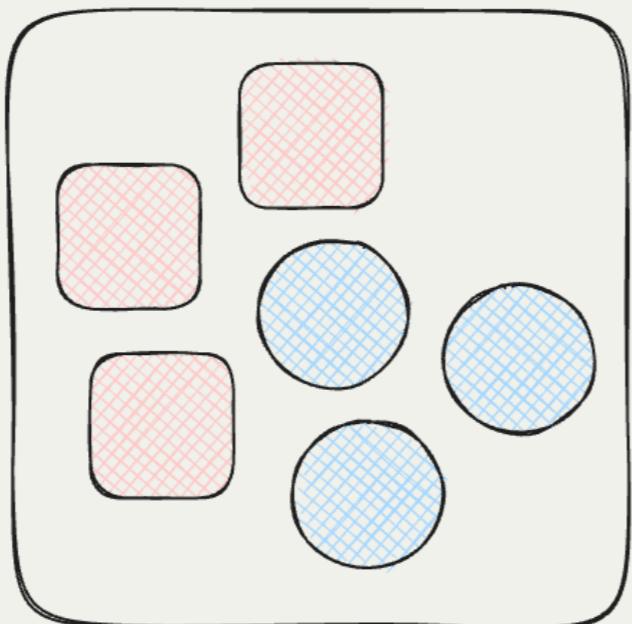
```
1 trait Area {  
2     def area: Double  
3 }  
4  
5 class Circle(radius: Double) extends Area {  
6     override def area: Double = math.Pi * math.pow(radius, 2)  
7 }  
8  
9 class Rectangle(width: Double, length: Double) extends Area {  
10    override def area: Double = width * length  
11 }
```

```
1 // Обобщенная функция  
2 def areaOf(area: Area): Double = area.area  
3  
4 areaOf(new Circle(10))  
5 areaOf(new Rectangle(5, 5))
```

Сущности, которые представляют данные

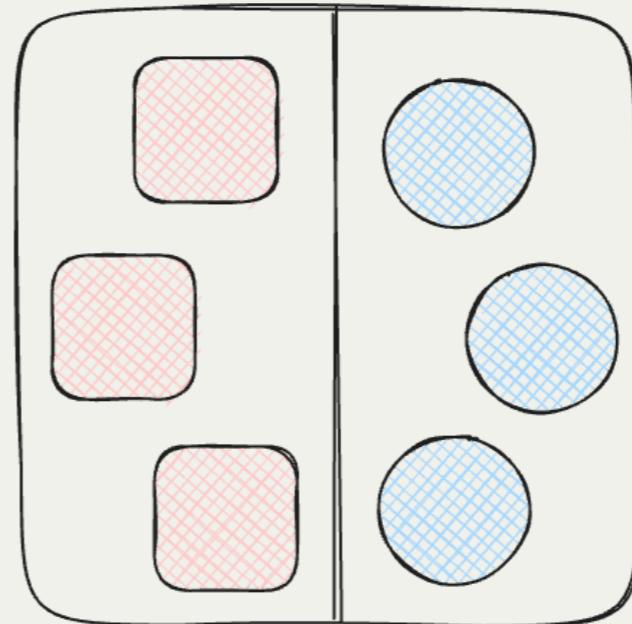
Сущности, которые представляют поведение

Полиморфизм через наследование



Вместе

Полиморфизм через тайпклассы



Отделены

Полиморфизм через тайпклассы

```
1 // сущности, представляющие данные
2 case class Circle(radius: Double)
3 case class Rectangle(width: Double, length: Double)
```

Полиморфизм через тайпклассы

```
1 // сущности, представляющие данные
2 case class Circle(radius: Double)
3 case class Rectangle(width: Double, length: Double)
```

```
1 // тайпкласс
2 trait Area[A] {
3   def area(a: A): Double
4 }
5
6 // сущности, отвечающие за реализацию
7 object CircleArea extends Area[Circle] {
8   override def area(circle: Circle): Double = math.Pi * math.pow(circle.radius, 2)
9 }
10
11 object RectangleArea extends Area[Rectangle] {
12   override def area(rectangle: Rectangle): Double = rectangle.width * rectangle.length
13 }
```

Полиморфизм через тайпклассы

```
1 // сущности, представляющие данные
2 case class Circle(radius: Double)
3 case class Rectangle(width: Double, length: Double)
```

```
1 // тайпкласс
2 trait Area[A] {
3   def area(a: A): Double
4 }
5
6 // сущности, отвечающие за реализацию
7 object CircleArea extends Area[Circle] {
8   override def area(circle: Circle) : Double = math.Pi * math.pow(circle.radius, 2)
9 }
10
11 object RectangleArea extends Area[Rectangle] {
12   override def area(rectangle: Rectangle): Double = rectangle.width * rectangle.length
13 }
```

```
1 // Обобщенная функция
2 def areaOf[A](shape: A, area: Area[A]): Double = area.area(shape)
3
4 areaOf(Circle(42), CircleArea)
5 areaOf(Rectangle(12, 15), RectangleArea)
```

Полиморфизм через тайпклассы

```
1 // Обобщенная функция
2 def areaOf[A](shape: A, area: Area[A]): Double = area.area(shape)
3
4 // Слишком много кода
5 areaOf(Circle(42), new CircleArea)
6 areaOf(Rectangle(12, 15), new RectangleArea)
```

Полиморфизм через тайпклассы

```
1 // Инстансы тайплкассов
2 implicit val circleArea: Area[Circle] = new Area[Circle] {
3   override def area(circle: Circle): Double = math.Pi * math.pow(circle.radius, 2)
4 }
5
6 implicit val rectangleArea: Area[Rectangle] = new Area[Rectangle] {
7   override def area(rectangle: Rectangle): Double = rectangle.width * rectangle.length
8 }
```

```
1 // Обобщенная функция
2 def areaOf[A](figure: A)(implicit area: Area[A]): Double = area.area(figure)
3
4 areaOf(Circle(42))
5 areaOf(Rectangle(12, 15))
```

Полиморфизм через тайпклассы

```
1 // Синтаксис для тайпкласса Area
2 implicit class AreaSyntax[A](val figure: A) extends AnyVal {
3   def area(implicit area: Area[A]): Double = area.area(figure)
4 }
5
6 Circle(42).area
7 Rectangle(12, 15).area
```

- Тайпклассы можно реализовать не только в Scala
- За счет имплиситов тайпклассы в Scala выглядят более выразительно

Анатомия тайпклассов

- trait (сам тайпкласс)
- методы тайпклассов
- инстансы тайпклассов
- implicit class - синтаксис (опционально)

Анатомия тайпклассов

```
// тайпкласс
trait TypeClass[A] {
    def method(value: A): Unit
}
```

Анатомия тайпклассов

```
1 // Инстансы
2 implicit val intInstance: TypeClass[Int] = new TypeClass[Int] {
3     def method(value: Int): Unit = ???
4 }
5
6 implicit val intInstance: TypeClass[String] = new TypeClass[String] {
7     def method(value: String): Unit = ???
8 }
```

Анатомия тайпклассов

```
1 // Синтаксис
2 implicit class TypeClassOps[A](private val value: A) extends AnyVal {
3     def method(implicit ev: TypeClass[A]): Unit = ev.method(value)
4 }
```

Использование тайпкласса

```
1 object SomeApp {  
2     def someMethod[A](arg: A)(implicit t: TypeClass[A]): Unit = {  
3         t.method(arg)  
4     }  
5 }
```

Использование тайпкласса

```
1 trait TypeClass[A] {  
2     def method(value: A): Unit  
3 }  
4  
5 // создаем объект с методом apply  
6 object TypeClass {  
7     def apply[A](implicit ev: TypeClass[A]): TypeClass[A] = ev  
8 }
```

```
1 object SomeApp {  
2     def someMethod[A: TypeClass](arg: A): Unit = {  
3         // достаем инстанс через apply  
4         TypeClass[A].method(arg)  
5     }  
6 }
```

Использование синтаксиса

```
1 object SomeApp {  
2     import TypeClassSyntax._  
3  
4     def someMethod[A: TypeClass](arg: A): Unit = {  
5         arg.method  
6     }  
7 }
```

Вопросы?

Simple type classes

Show

Show - альтернатива Java `toString`.

Show

Что не так с `.toString()`?

```
(new {}).toString  
// res0: String = "repl.MdocSession`$MdocApp$`anon$1@7de74c88"
```

`toString` определен для `Object` и может быть вызван для чего угодно

Show

Show позволяет нам определять преобразования в строковое представление только для тех типов данных, которые нам действительно нужны.

Show

```
1 // тайпкласс
2 trait Show[A] {
3     def show(value: A): String
4 }
5
6 // объект-компаньон с методом apply
7 object Show {
8     def apply[A](implicit ev: Show[A]): Show[A] = ev
9 }
```

Show

```
1 // инстансы для Int и String
2 object ShowInstances {
3     implicit val showInt = new Show[Int] {
4         def show(value: Int): String = value.toString
5     }
6
7     implicit val showString = new Show[String] {
8         def show(value: String): String = value
9     }
10 }
```

Show

```
1 // синтаксис
2 object ShowSyntax {
3     implicit class ShowOps[A](private val value: A) extends AnyVal {
4         def show(implicit ev: Show[A]): Unit = ev.show(value)
5     }
6 }
```

Show

Пример использования с примитивными типами

```
1 import ShowInstances._  
2 import ShowSyntax._  
3  
4 val meaningOfLife = 42  
5  
6 Show[ Int ].show(meaningOfLife)    // result: "42"  
7 // OR  
8 meaningOfLife.show    // result: "42"
```

Show

Пример использования с кастомными типами

```
1 import ShowInstances._  
2 import ShowSyntax._  
3  
4 case class User(name: String, age: Int)  
5  
6 object User {  
7     implicit val showUser = new Show[User] {  
8         def show(user: User): String =  
9             s"User(name = `${user.name}, age = ${user.age})"  
10    }  
11 }  
12  
13 val user = User("Mark", 25)  
14 user.show // result: "User(name = Mark, age = 25)"
```

Eq

```
1 trait Eq[A] {  
2   def eqv(x: A, y: A): Boolean  
3 }
```

Eq является альтернативой Java equals

Eq

Что не так с Java equals?

Eq

Что не так с Java equals?

```
"Hello" == 42
```

Eq

```
1 // Тайпкласс
2 trait Eq[A] {
3   def eqv(x: A, y: A): Boolean
4 }
5
6 // Объект компаньон с методом apply
7 object Eq {
8   def apply[A](implicit ev: Eq[A]): Eq[A] = ev
9 }
```

Eq

```
1 // инстансы для Int и String
2 object EqInstances {
3     implicit val eqInt = new Eq[Int] {
4         def eqv(x: Int, y: Int): Boolean = x == y
5     }
6
7     implicit val eqString = new Eq[String] {
8         def eqv(x: String, y: String): Boolean = x == y
9     }
10 }
```

Eq

```
1 // Синтаксис
2 object EqSyntax {
3     implicit class EqOps[A](private val x: A) extends AnyVal {
4
5         def eqv(y: A)(implicit ev: Eq[A]): Unit = ev.eqv(x, y)
6
7         def ===(y: A)(implicit ev: Eq[A]): Unit = ev.eqv(x, y)
8
9         def !==(y: A)(implicit ev: Eq[A]): Unit = !ev.eqv(x, y)
10    }
11 }
```

Eq

Пример использования с примитивными типами

```
1 import EqInstances._  
2 import EqSyntax._  
3  
4 Eq[Int].eqv(2 + 2, 4)    // result: true  
5  
6 "Hello" === "world"      // result: false  
7 "Hello" =!= "world"      // result: true
```

Eq

Пример использования с примитивными типами

```
1 import EqInstances._  
2 import EqSyntax._  
3  
4 "Hello" === 42  
5 /*  
6  [error]  App.scala:202:15: type mismatch;  
7  [error]    found   : Int(42)  
8  [error]    required: String  
9  [error]      "Hello" === 42  
10 [error]                  ^  
11 [error] one error found  
12 [error] (Compile / compileIncremental) Compilation failed */
```

Eq

Пример использования с кастомными типами

```
1 case class User(name: String, age: Int)
2
3 object User {
4     implicit val eqUser = new Eq[User] {
5         def eqv(x: User, y: User): Boolean = {
6             x.name === y.name && x.age === y.age
7         }
8     }
9 }
10
11 val mark = User("Mark", 25)
12 val joe = User("Joe", 33)
13
14 mark === joe // result: false
```

Сегодня рассмотрели:

- Implicit conversions
- Implicit parameters
- Implicit classes
- Type classes
- Simple type classes

Спасибо за внимание!

