

Scala School

Лекция 2: Основы (продолжение)

BINARYDISTRICT

План лекции

- OOP: Class, trait, object, enum
- Case class
- Сопоставление с образцом
- Обработка исключений
- Either
- Формы вызова методов



OOP: Classes, traits, objects, enums

OOP

- `class`
- `abstract class`
- `trait`
- `object`
- `enum`

Class

```
class Person(firstName: String, lastName: String, age: Int) {  
    var money: Double = 0  
  
    def this(firstName: String, lastName: String) = {  
        this(firstName, lastName, 0)  
    }  
}
```

```
val john = new Person("John", "Smith", 25)  
val mike = new Person("Mike", "Johnson")
```

Class constructor

```
class Foo(a: Int, val b: Int, var c: Int) { def sum = a + b + c }
```

```
object FooTest extends App {  
    val x = new Foo(1, 2, 3)  
    // Нельзя вызвать x.a  
    println(x.b) // 2  
    println(x.c) // 3  
    println(x.sum) // 6  
    x.c = 4  
    println(x.sum) // 7  
}
```

Class constructor - формы записи поля

`a: Int` - создает `private` поле `a`

`val b: Int` - создает `private` поле `b` и `getter`

`var c: Int` - создает `private` поле `c`, `getter` и `setter`

Abstract class

```
abstract class Vehicle (brand: String, maxSpeed: Double) {  
    val numWheels: Int  
    def ride(): Unit  
    def stop(): Unit = println("Stop!")  
}
```

```
class Car (brand: String, maxSpeed: Double, numSeats: Int) extends  
Vehicle (brand, maxSpeed) {  
    override val numWheels: Int = 4  
    override def ride(): Unit = println("wrrrooom")  
}
```


Trait

Trait - аналог интерфейса в Java

- имплементация полей и методов
- множественное наследование

Trait

```
trait VehicleLike {  
    val numWheels: Int  
    def ride(): Unit  
    def stop(): Unit = println("Stop!")  
    val brand: String  
    val maxSpeed: Double  
}  
  
class Moto(val brand: String, val maxSpeed: Double) extends VehicleLike {  
    override val numWheels: Int = 2  
    override def ride(): Unit = println("Woom")  
}
```

Trait: множественное наследование

```
trait Cargo {  
    def carryStuff(s: Any): Unit = println(s"$s is on board!")  
}
```

```
class Truck(val brand: String, val maxSpeed: Double) extends VehicleLike with  
Cargo {  
    override val numWheels: Int = 4  
    override def ride(): Unit = println("Wrrrrr")  
}
```

Trait: множественное наследование

```
trait RunOne { def run(): Unit = println("Run") }  
trait RunTwo { def run(): Unit = println("Run, Forest!") }  
  
class GoodRunner extends RunOne with RunTwo
```

Trait: множественное наследование

```
trait RunOne { def run(): Unit = println("Run") }
```

```
trait RunT
```

```
class Good
```



Trait: множественное наследование

```
trait RunOne { def run(): Unit = println("Run") }  
trait RunTwo { def run(): Unit = println("Run, Forest!") }
```

```
class GoodRunner extends RunOne with RunTwo
```

error: *class GoodRunner inherits conflicting members:*

method run in trait RunOne of type ()Unit and

method run in trait RunTwo of type ()Unit

(Note: this can be resolved by declaring an override in class GoodRunner.)

Trait: множественное наследование

```
trait RunOne { def run(): Unit = println("Run") }  
trait RunTwo { def run(): Unit = println("Run, Forest!") }  
  
class GoodRunner extends RunOne with RunTwo {  
  override def run(): Unit = println("Run, Forest, run!")  
}
```

Trait vs Abstract class

- `trait (interface)` - отвечает на вопрос “что делает”?
- `abstract class` - “чем является?”

Object

В Scala нет ключевого слова `static`, но есть сущность `object` - синглтон

```
object MyObject {  
    val x = "scala"  
    def foo(str: String) = s"hi, $str"  
}  
  
val res = MyObject.foo(MyObject.x) // hi, scala
```

`object` может наследовать классы и трейты

```
object InheritedObject extends Serializable { /* ... */ }
```

Companion Object

Часто для класса создается одноименный `object` - companion object

```
object Person {  
    private val hello = "Hello, "  
    def apply(firstName: String, lastName: String, age: Int): Person =  
        new Person(firstName, lastName, age)  
}  
  
class Person(firstName: String, lastName: String, age: Int) {  
    import Person._  
    def greet(question: String) = hello + firstName + "!" + question  
}
```

Package object

- Хранит переменные и методы, по умолчанию доступные для всего пакета
- Можно импортировать и использовать в других пакетах

```
import wtf.scala.lectures.eo2._
```

Package object

```
// file wtf.scala.lectures.eo2.package.scala
package wtf.scala.lectures

package object e02 {
    val VersionString = "1.0"
    type JList[T] = java.util.List[T]
    def doSmtH(str: String): String = // ...
    implicit def a2b(a: A): B = // ...
}
```

Sealed

Ключевое слово `sealed` в объявлении класса (трейта) означает, что его прямые наследники могут быть объявлены только в этом же файле.

Наследники наследников могут быть объявлены в других файлах

Sealed

Ключевое слово `sealed` в объявлении класса (трейта) означает, что его прямые наследники могут быть объявлены только в этом же файле.

Наследники наследников могут быть объявлены в других файлах

Примеры:

- `scala.Option`
- `scala.util.Try`
- `scala.util.Either`

Enum

В Scala нет отдельной структуры enum.

Вместо этого создается `object`, наследующий `scala Enumeration`.

Enum: Java vs Scala

Java:

```
public enum Season {  
    WINTER, SPRING, SUMMER, AUTUMN  
}
```

Scala:

```
object Season extends Enumeration {  
    val Winter, Spring, Summer,  
    Autumn = Value  
}
```


Enum: Java vs Scala

Java:

```
Season.valueOf(String name)
```

```
Season.values()
```

Scala:

```
Season.withName(name: String)
```

```
Season.values
```

```
Season.apply(x: Int)
```

```
Season.maxId
```

Enum: Java vs Scala

Java:

```
Season.WINTER.ordinal()
```

```
Season.WINTER.compareTo(Season  
that)
```

Scala:

```
Season.Winter.id
```

```
Season.Winter.compareTo(that:  
Season)
```

```
Season.Winter + Season.Spring
```

```
Season.Winter < Season.Spring
```

Case class

Метод apply

Синтаксический сахар:

```
object Applicable {  
    def apply(number: Int): String = s"My number is $number"  
}
```

```
scala> Applicable.apply(7)  
res0: String = My number is 7
```

```
scala> Applicable(7)  
res1: String = My number is 7
```

scala.Product



Базовый трейт, используемый для представления объектов - кортежей

```
trait Product extends Any with Equals {  
  def productElement (n: Int): Any  
  def productArity: Int  
  def productIterator: Iterator[Any] = /* implementation */  
}
```

Наследники: `Product1`, `Product2`, ..., `Product22`

Case class

case class - класс со следующими свойствами:

- Автогенерируемый companion object с методами `apply`, `unapply`
- Автогенерируемые `toString()`, `equals`, `hashCode()`
- Автогенерируемый `copy`

Case class

case class - класс со следующими свойствами:

- Автогенерируемый companion object с методами `apply`, `unapply`
- Автогенерируемые `toString()`, `equals`, `hashCode()`
- Автогенерируемый `copy`
- Все поля конструктора - `val (immutable)`
- Наследует трейты `Product` и `Serializable`

Case class

case class - класс со следующими свойствами:

- Автогенерируемый companion object с методами `apply`, `unapply`
- Автогенерируемые `toString()`, `equals`, `hashCode()`
- Автогенерируемый `copy`
- Все поля конструктора - `val` (immutable)
- Наследует трейты `Product` и `Serializable`

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

```
scala> Person("Ivan", 27)
```

```
res1: Person = Person(Ivan,27)
```


Case class: copy

```
scala> val ivan = Person("Ivan", 27)
```

```
ivan: Person = Person(Ivan,27)
```

```
scala> val john = ivan.copy(name = "John", age = 24)
```

```
john: Person = Person(John,24)
```

```
scala> val peter = ivan.copy(name = "Peter")
```

```
peter: Person = Person(Peter,27)
```

Case class: примеры

- Tuple
- Some / None
- Success / Failure
- ...

Case object

Синглтон case class без аргументов конструктора

- Обладает всеми свойствами case class
- Часто используется в Akka
- case class без аргументов конструктора - deprecated

```
case object CaseObject
```

Tuple

`Tuple1`, ..., `Tuple22` - кортеж из нескольких элементов

```
val tuple: (Int, String, Double) = (1, "Scala", 2.12) // Синтаксический сахар  
  
val first = tuple._1 // 1  
  
val (a, b) = (true, "Scala")
```

Tuple

`Tuple1`, ..., `Tuple22` - кортеж из нескольких элементов

```
val tuple: (Int, String, Double) = (1, "Scala", 2.12) // Синтаксический сахар
```

```
val first = tuple._1 // 1
```

```
val (a, b) = (true, "Scala")
```

```
final case class Tuple3[+T1, +T2, +T3](_1: T1, _2: T2, _3: T3)
```

```
  extends Product3[T1, T2, T3]
```

```
{
```

```
  override def toString() = "(" + _1 + "," + _2 + "," + _3 + ")"
```

```
}
```

Монада

Контейнер для объектов, применяется для связывания вычислений

- `apply` - создание новой монады из объекта
- `flatMap` - создание новой монады, содержащий результат функции от содержимого исходной монады

Монада

Контейнер для объектов, применяется для связывания вычислений

- `apply` - создание новой монады из объекта
- `flatMap` - создание новой монады, содержащий результат функции от содержимого исходной монады

```
trait Monad[T] {  
    def flatMap[U] (f: T => Monad[U]): Monad[U]  
}
```

```
def apply[T] (x: T): Monad[T]
```

Монада

Часто используемым методом монад является `map`,
который можно получить комбинацией методов `flatMap` и `unit`

```
def map[U] (f: T => U): Monad[U] = {  
    flatMap { (x: T) =>  
        Monad.apply(f(x))  
    }  
}
```


Монада

Примеры:

- `scala.Option`
- `scala.util.Try`
- `scala.util.Either`
- `List`
- `...`

Option

`scala.Option[T]` - контейнер, задача которого - содержать опциональное значение, которого может не быть

- `Some (x)` - хранит значение `x` объекта
- `None` - объект отсутствует

Option

`scala.Option[T]` - контейнер, задача которого - содержать опциональное значение, которого может не быть

- `Some(x)` - хранит значение `x` объекта
- `None` - объект отсутствует

```
sealed abstract class Option[+A] extends Product with Serializable
```

```
final case class Some[+A](value: A) extends Option[A]
```

```
case object None extends Option[Nothing]
```

Создание Option

```
val someMessage: Option[String] = Some("Hello!")
```

```
val noneMessage: Option[String] = None
```

```
val existingMessage: Option[String] = Option("Hi") // Some("hi")
```

```
val absentMessage: Option[String] = Option(null) // None
```

Option: методы

```
def isEmpty: Boolean
```

```
def get: A
```

```
def getOrElse[B >: A](default => B): B
```

Option: методы

```
val someString = Some("Wow")
```

```
scala> someString.isEmpty
```

```
res2: Boolean = false
```

```
val noneString: Option[String] = None
```

```
scala> noneString.isEmpty
```

```
res5: Boolean = true
```

Option: методы

```
val someString = Some("Wow")
```

```
scala> someString.isEmpty
```

```
res2: Boolean = false
```

```
scala> someString.get
```

```
res3: String = Wow
```

```
val noneString: Option[String] = None
```

```
scala> noneString.isEmpty
```

```
res5: Boolean = true
```

```
scala> noneString.get
```

```
java.util.NoSuchElementException: None.get
```

Option: методы

```
val someString = Some("Wow")
```

```
scala> someString.isEmpty
```

```
res2: Boolean = false
```

```
scala> someString.get
```

```
res3: String = Wow
```

```
scala> someString.getOrElse("Nope")
```

```
res4: String = Wow
```

```
val noneString: Option[String] = None
```

```
scala> noneString.isEmpty
```

```
res5: Boolean = true
```

```
scala> noneString.get
```

```
java.util.NoSuchElementException: None.get
```

```
scala> noneString.getOrElse("Nope")
```

```
res7: String = Nope
```


Option: map, flatMap

```
def map[B] (f: A => B): Option[B] =  
    if (isEmpty) None else Some(f(this.get))
```

```
def flatMap[B] (f: A => Option[B]): Option[B] =  
    if (isEmpty) None else f(this.get)
```

Option: map, flatMap

```
def map[B] (f: A => B): Option[B] =  
    if (isEmpty) None else Some(f(this.get))
```

```
def flatMap[B] (f: A => Option[B]): Option[B] =  
    if (isEmpty) None else f(this.get)
```

```
scala> Some("Scala").map(s: String) => s.toLowerCase
```

```
res5: Option[String] = Some(scala)
```

```
scala> None.map((s: String) => s.toLowerCase)
```

```
res6: Option[String] = None
```

Option: for comprehensions

```
object OptionForComprehensions {  
  val nameOpt = Some("Ivan")  
  val lastNameOpt = Some("Petrov")  
  
  val x = for {  
    name <- nameOpt  
    lastName <- lastNameOpt  
  } yield name + lastName // x = ???  
  
}
```

Option: for comprehensions

```
val x = for {  
  name <- nameOpt  
  lastName <- lastNameOpt  
} yield name + lastName
```

```
val y = nameOpt.flatMap {  
  name => lastNameOpt.map {  
    lastName => name + lastName  
  }  
}
```

Option: for comprehensions

```
val x = for {  
  name <- nameOpt  
  lastName <- lastNameOpt  
} yield name + lastName // Some("IvanPetrov")
```

```
val y = nameOpt.flatMap {  
  name => lastNameOpt.map {  
    lastName => name + lastName  
  }  
} // Some("IvanPetrov")
```

Option: for comprehensions

```
object OptionForComprehensions {  
  val nameOpt = Some("Ivan")  
  val lastNameOpt: Option[String] = None  
  
  val x = for {  
    name <- nameOpt  
    lastName <- lastNameOpt  
  } yield name + lastName // None  
  
}
```

Option: еще методы

```
def contains[A1 >: A](elem: A1): Boolean
```

```
def exists(p: A => Boolean): Boolean
```

```
def forall(p: A => Boolean): Boolean
```

```
def foreach[U](f: A => U): Unit
```

```
def collect[B](pf: PartialFunction[A, B]): Option[B]
```

```
def orElse[B >: A](alternative: => Option[B]): Option[B]
```

```
def toList: List[A]
```

```
...
```

Сопоставление с образцом



Сопоставление с образцом

Сопоставление с образцом (Pattern matching) - одна из наиболее часто используемых возможностей Scala

```
def matchInt (num: Int): String = {  
    num match {  
        case 1 => "one"  
        case 2 => "two"  
        case _ => "many" // все остальное  
    }  
}
```

Сопоставление с образцом

Можно использовать if condition

```
def matchIntCondition (num: Int): String = {  
  num match {  
    case i if i == 1 => "one"  
    case i if i == 2 | i == 3 => "two or three"  
    case _ => "many"  
  }  
}
```

Сопоставление с образцом

Сопоставление по типу

```
def matchType(obj: Any): String = {  
  obj match {  
    case 1 => "one"  
    case "two" => "two"  
    case y: Int => "many"  
    case z @ _ => s"other: $z" // все остальное  
  }  
}
```

Сопоставление с образцом

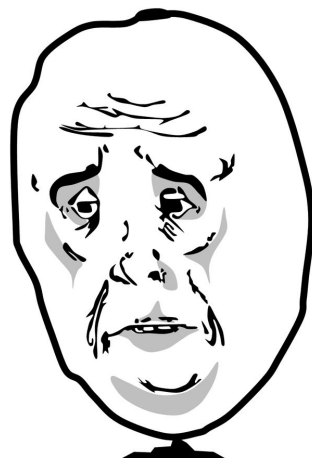
А если что-то пропустили?

```
def matchNotExhaustive (num: Int): String = {  
  num match {  
    case 1 => "one"  
    case 2 => "two"  
  }  
}
```

Сопоставление с образцом

А если что-то пропустили?

```
def matchNotExhaustive (num: Int): String = {  
  num match {  
    case 1 => "one"  
    case 2 => "two"  
  }  
}  
  
scala> matchNotExhaustive(5)  
  
scala.MatchError: 5 (of class java.lang.Integer)
```



Сопоставление с образцом: case class

Для case class возможно более сложное сравнение с образцом - по значению конкретных полей

```
object CaseClassMatching extends App {  
  val opt: Option[Int] = Some(4)  
  opt match {  
    case Some(n) if n > 5 => println(s"My number $n > 5")  
    case Some(_) => println("My number is <5") // matched  
    case None => println("No number")  
  }  
}
```

Сопоставление с образцом: extractors, unapply

Для сопоставления с образцом не обязательно использовать case class.

Пример: `scala.util.control.NonFatal`

Для этого необходим object с методом `unapply`, этот object называется экстрактором (extractor object).

Сопоставление с образцом: extractors, unapply

Для сопоставления с образцом не обязательно использовать case class.

Пример: `scala.util.control.NonFatal`

Для этого необходим `object` с методом `unapply`, этот `object` называется экстрактором (extractor object).

В case class метод `unapply` в companion object генерируется автоматически, также, так и метод `apply`.

Сопоставление с образцом: extractors, unapply

```
object Twice {  
    def unapply(num: Int): Option[Int] = if (num % 2 == 0) Some(num/2) else  
None  
}
```

```
object TwiceExample extends App {  
    val x = 34  
    x match {  
        case Twice(n) => println(s"Twice $n") // Twice 17  
        case _ => println("Not twice")  
    }  
}
```

Сопоставление с образцом: extractors, unapply

Что произошло?

- `case Twice(n)` – вызывает метод `Twice.unapply(x)`

Сопоставление с образцом: `extractors`, `unapply`

Что произошло?

- `case Twice(n)` – вызывает метод `Twice.unapply(x)`
- Если результат выполнения - `Some(n)`, значит, образец подошел и значение, содержащееся в `Option` (в нашем случае, `num / 2`), возвращается

Сопоставление с образцом: `extractors`, `unapply`

Что произошло?

- `case Twice(n)` – вызывает метод `Twice.unapply(x)`
- Если результат выполнения - `Some(n)`, значит, образец подошел и значение, содержащееся в `Option` (в нашем случае, `num / 2`), возвращается
- Если результат `None`, образец не подошел

Сопоставление с образцом: `extractors`, `unapply`

Что должен возвращать метод `unapply`?

- Просто проверить условие - `Boolean`
- Вернуть одно значение типа `T` - `Option[T]`
- Вернуть фиксированное число значений - `Option[(T1, ..., Tn)]`

Сопоставление с образцом

```
object Even {  
    def unapply(num: Int): Boolean = num % 2 == 0  
}
```

```
object EvenExample extends App {  
    4 match {  
        case e @ Even() => println(s"$e is even") // 4 is even  
        case e @ _ => println(s"$e is odd")  
    }  
}
```

Сопоставление с образцом: unapplySeq

Если число значений заранее неизвестно, используется метод `unapplySeq`

```
object Domain {  
  def unapplySeq(dom: String): Option[Seq[String]] = Some(dom.split("\\\\"))  
}
```

Сопоставление с образцом: unapplySeq

Если число значений заранее неизвестно, используется метод `unapplySeq`

```
object Domain {  
  def unapplySeq(dom: String): Option[Seq[String]] = Some(dom.split("\\\\"))  
}  
  
object DomainExample extends App {  
  "google.com" match {  
    case Domain("google", "com") => println("Hey, Google!") // matches  
    case Domain("yahoo", _) => println("yahoo")  
    case _ => println("other")  
  }  
}
```


Обработка исключений

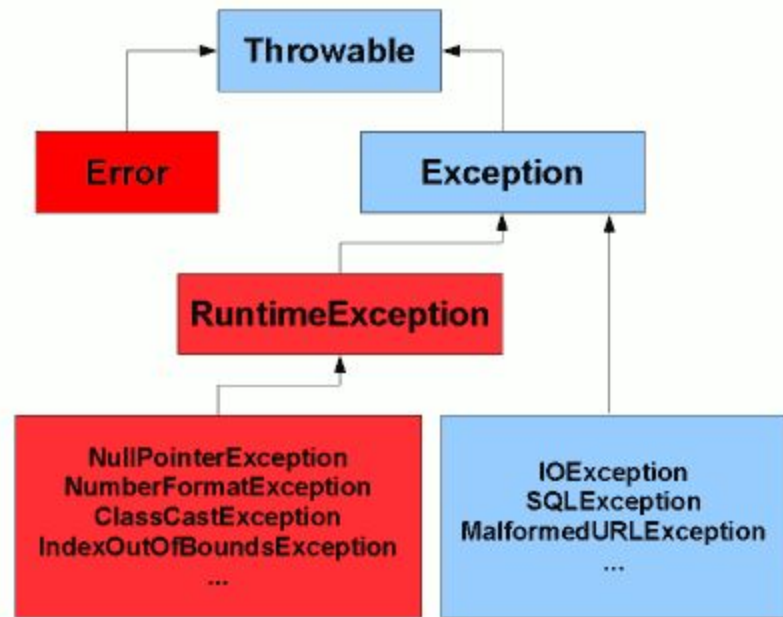
Выбрасывание исключений

В Scala, как и в Java, используется ключевое слово `throw`

```
def checkMoney (money: Double): Unit = {  
    if (money <= 0) {  
        throw new RuntimeException ("Insufficient funds")  
    }  
}
```

Иерархия исключений

- В Scala используются Java - исключения
- В отличие от Java, unchecked - исключения не обязательно перехватывать или указывать в сигнатуре метода



Обработка исключений: Java vs Scala

```
try {  
    // Something dangerous  
} catch (Exception e) {  
    e.printStackTrace();  
} finally {  
    System.out.println("Next");  
}
```

```
try {  
    // Something dangerous  
} catch {  
    case ex: Exception =>  
        ex.printStackTrace()  
} finally {  
    println("Next")  
}
```

Обработка исключений в Scala

В Scala try / catch блок возвращает результат

```
val x = try {  
    Integer.parseInt("10")  
} catch {  
    case ex: Exception =>  
        0  
}
```

Обработка исключений в Scala

Блок `catch` принимает `PartialFunction[Throwable, Any]`

```
def handler: PartialFunction[Throwable, Any] = {  
  case iae: IllegalArgumentException =>  
    System.out.println("Got iae")  
  case ex: RuntimeException =>  
    System.out.println("Got ex" + ex.getMessage)  
}  
  
try {  
  // do something  
} catch handler
```

Try, Success, Failure

`scala.util.Try[T]` – функциональный способ обработки исключений

Имеет 2 наследников:

- `Success(x)` - хранит результат `x` успешной операции
- `Failure(ex)` - хранит исключение `ex`, если операция не успешна

Try, Success, Failure

`scala.util.Try[T]` – функциональный способ обработки исключений

Имеет 2 наследников:

- `Success(x)` - хранит результат `x` успешной операции
- `Failure(ex)` - хранит исключение `ex`, если операция не успешна

```
Try(Integer.parseInt("1")) match {  
  case Success(number) => println(number)  
  case Failure(ex)    => ex.printStackTrace()  
}
```


Try

```
sealed abstract class Try[+T] extends Product with Serializable {  
  def isFailure: Boolean  
  def isSuccess: Boolean  
  def toOption: Option[T]  
  def flatMap[U](f: T => Try[U]): Try[U]  
  def map[U](f: T => U): Try[U]  
}
```

```
final case class Success[+T](value: T) extends Try[T]  
final case class Failure[+T](exception: Throwable) extends Try[T]
```

Try

```
object TryExample {  
    val success = Try(Integer.parseInt("1"))  
    val failure = Try(Integer.parseInt("oops"))  
  
    success.map(i => i + 1) // Success(2)  
    failure.map(i => i + 1) // Failure(java.lang.NumberFormatException: For  
input string: "oops")  
  
}
```

Try

```
object TryExample {  
    val success = Try(Integer.parseInt("1"))  
    val failure = Try(Integer.parseInt("oops"))  
  
    success.map(i => i + 1) // Success(2)  
    failure.map(i => i + 1) // Failure(java.lang.NumberFormatException: For  
input string: "oops")  
  
    success.toOption // Some(1)  
    failure.toOption // None  
}
```

Either

Either

Класс `scala.util.Either[A, B]` - монада, содержащая значение, принадлежащее одному из двух возможных непересекающихся типов

Either

Класс `scala.util.Either[A, B]` - монада, содержащая значение, принадлежащее одному из двух возможных типов

```
sealed abstract class Either[+A, +B] extends Product with Serializable
```

```
final case class Right[+A, +B](value: B) extends Either[A, B]
```

```
final case class Left[+A, +B](value: A) extends Either[A, B]
```

Усложненный `Option`: `Some` -> `Right`, `None` -> `Left`

Either - пример

```
val in = Console.readLine("Type Either a string or an Int: ")
val result: Either[String,Int] = try {
    Right(in.toInt)
} catch {
    case e: Exception =>
        Left(in)
}
println(result match {
    case Right(x) => s"You passed Int: $x"
    case Left(x)  => s"You passed String: $x"
}))
```

Either

```
def isLeft: Boolean
```

```
def isRight: Boolean
```

```
// Right-biased since scala 2.12, Left-biased before
```

```
def flatMap[AA >: A, Y](f: B => Either[AA, Y]): Either[AA, Y] = this match  
{  
  case Right(b) => f(b)  
  case Left(a)  => this.asInstanceOf[Either[AA, Y]]  
}
```

```
def left = Either.LeftProjection(this) // To process Left
```

```
def right = Either.RightProjection(this) // Compatibility with <= 2.11
```


Either

```
object EitherExample {  
    val right: Either[String, Int] = Right(10)  
    val left: Either[String, Int] = Left("LEFT")  
  
    val x = right.map(s => s "I've got $s") // Right(I've got 10)  
    val y = left.map(s => s "I've got $s")  // Left(LEFT)  
    val z = left.left.map(s => s "I've got $s") // Left(I've got LEFT)  
}
```

Формы вызова методов

Префиксная форма вызова метода

```
case class Unary(num: Int) { /* ??? */ }
```

```
object UnaryTest {  
    val plus = +Unary(3) // Unary(4)  
}
```

Префиксная форма вызова метода

```
case class Unary(num: Int) { /* ??? */ }
```

```
object UnaryTest {  
    val plus = +Unary(3) // Unary(4)  
}
```

В Scala можно определить 4 префиксных метода: +, -, !, ~

Для этого необходимо задать метод без аргументов

```
unary_+(), unary_-(), unary_!(), unary~()
```

Префиксная форма вызова метода

```
case class Unary(num: Int) {  
  def unary_+() = Unary(num + 1)  
  def unary_-() = Unary(num - 1)  
  def unary_!() = Unary(num * 2)  
  def unary_~() = Unary(num * num)  
}
```

Префиксная форма вызова метода

```
case class Unary(num: Int) {  
    def unary_+() = Unary(num + 1)  
    def unary_-() = Unary(num - 1)  
    def unary_!() = Unary(num * 2)  
    def unary_~() = Unary(num * num)  
}
```

```
object UnaryTest {  
    val plus = +Unary(3) // Unary(4)  
    val minus = -Unary(3) // Unary(2)  
    val bang = !Unary(3) // Unary(6)  
    val tilde = ~Unary(3) // Unary(9)  
  
    // не рекомендуется  
    val badPlus = Unary(3).unary_+()  
}
```

Инфиксная форма вызова метода

В Scala методы одного аргумента можно записывать в инфиксной форме:

- без использования точки перед именем метода
- без использования скобок вокруг аргумента

```
val str = "hello"  
str.endsWith("o")  
str endsWith "o"
```

Порядок вызова инфиксных методов

```
case class Infix(num: Int) {  
  def +(that: Infix) = Infix(this.num + that.num)  
  def add(that: Infix) = Infix(this.num + that.num)  
  def *(that: Infix) = Infix(this.num * that.num)  
  def mult(that: Infix) = Infix(this.num * that.num)  
}
```


Порядок вызова инфиксных методов

```
case class Infix(num: Int) {  
  def +(that: Infix) = Infix(this.num + that.num)  
  def add(that: Infix) = Infix(this.num + that.num)  
  def *(that: Infix) = Infix(this.num * that.num)  
  def mult(that: Infix) = Infix(this.num * that.num)  
}  
  
object Infix {  
  val a = Infix(1) add Infix(2) mult Infix(3)  
  val b = Infix(1) + Infix(2) * Infix(3)  
}
```

Порядок вызова инфиксных методов

```
case class Infix(num: Int) {  
  def +(that: Infix) = Infix(this.num + that.num)  
  def add(that: Infix) = Infix(this.num + that.num)  
  def *(that: Infix) = Infix(this.num * that.num)  
  def mult(that: Infix) = Infix(this.num * that.num)  
}  
  
object Infix {  
  val a = Infix(1) add Infix(2) mult Infix(3) // Infix(9)  
  val b = Infix(1) + Infix(2) * Infix(3) // Infix(7)  
}
```

Порядок вызова инфиксных методов

Порядок вызова инфиксных методов определяется первой буквой их имени.

(Все буквы)

|

^

&

= !

< >

:

+ -

* / %

(Остальные специальные символы)

Возрастание приоритета

Ассоциативность бинарных операторов

- По умолчанию - левая ассоциативность (вызов слева направо)
- Операторы, заканчивающиеся на : - право ассоциативные
- Для право ассоциативного оператора аргументы меняются местами

Ассоциативность бинарных операторов

```
case class InfixRight (num: Int) {  
    def ++(that: InfixRight) = InfixRight (this.num + this.num * that.num)  
    def +:(that: InfixRight) = InfixRight (this.num + this.num * that.num)  
}  
  
object InfixRight {  
    val a = InfixRight (0) ++ InfixRight (1) ++ InfixRight (2)  
  
    val b = InfixRight (0) +: InfixRight (1) +: InfixRight (2)  
  
}
```

Ассоциативность бинарных операторов

```
case class InfixRight (num: Int) {  
    def ++(that: InfixRight) = InfixRight (this.num + this.num * that.num)  
    def +:(that: InfixRight) = InfixRight (this.num + this.num * that.num)  
}  
  
object InfixRight {  
    val a = InfixRight (0) ++ InfixRight (1) ++ InfixRight (2) // InfixRight (0)  
    //      (InfixRight (0) ++ InfixRight (1)) ++ InfixRight (2)  
    val b = InfixRight (0) +: InfixRight (1) +: InfixRight (2) // InfixRight (4)  
    //      InfixRight (0) +: (InfixRight (1) +: InfixRight (2))  
    //      (InfixRight (2) ++ InfixRight (1)) ++ InfixRight (0)  
}
```

Постфиксная форма вызова метода

Постфиксная форма - вызов без точки метода без аргументов

```
object Postfix {  
    val trimmed = "1234 ".trim  
    val trimmedP = "1234 " trim  
}
```



Постфиксная форма не рекомендуется к использованию

Scala style guide: “This style is unsafe, and should not be used.”

<http://docs.scala-lang.org/style/method-invocation.html#arity-0>

Summary

- Scala - полноценный **объектно-ориентированный** язык
- **Монады** - функциональные контейнеры
- **Pattern matching**
- **Обработка исключений** в функциональном стиле
- **Гибкий синтаксис**



Bonus: Trait в Java

```
trait Foo {  
    def bar = { println("bar!") }  
}  
  
// Скомпилируется примерно в следующее  
  
public interface Foo {  
    public void bar();  
}  
  
public abstract class Foo$class {  
    public static void bar(Foo self) { println("bar!"); }  
}
```

Trait в Java

```
class Baz extends Foo
```

```
// Скомпилируется в
```

```
public class Baz implements Foo {  
    public void bar() { Foo$class.bar(this); }  
}
```

Префиксная форма вызова метода (Java)

```
public static void main(String[] args) {  
    Unary u = new Unary(3);  
  
    Unary plus = u.unary_$plus();  
    Unary minus = u.unary_$minus();  
    Unary bang = u.unary_$bang();  
    Unary tilde = u.unary_$tilde();  
}
```

Вызов операторных методов в Java

```
public static void main(String[] args) {  
    Infix infix = new Infix(3);  
    Infix plus = infix.$plus(new Infix(4));  
}
```

Спасибо

BINARYDISTRICT
