

Знакомство с объектно-ориентированными возможностями языка Kotlin

Романов Владимир Юрьевич
МГУ им. М.В.Ломоносова, ф-т ВМК
romanov.rvy@yandex.ru

1.1. ОБЗОР БАЗОВЫХ ВОЗМОЖНОСТЕЙ ЯЗЫКА KOTLIN.

Точки входа в программу

Точек входа в программу не ограничено.

Пример точки входа без параметра:

```
fun main() {  
    println("Hello Kotlin!")  
}
```

Пример точки входа с параметром:

```
fun main(args: Array<String>) {  
    println("Hello Kotlin!")  
}
```

Массив - это обычный класс

Пакеты

- Описание пакета должно быть в начале файла.
- Не требуется соответствия структуры пакетов структуре папок в файловой системе.
- Файл программы может быть расположен в любом месте.
- Точки с запятой в конце строки не требуется

Пример объявления пакета:

package tool.geometry

Импорты

Импортироваться могут классы, переменные, функции, ...

Импортируемые элементы могут получить псевдонимы.

Например, ***PI_From_Kotlin*** или ***E_From_Java***:

```
import java.lang.Math
```

```
import java.lang.Math.E as E_From_Java
```

```
import kotlin.math.PI as PI_From_Kotlin
```

```
fun main(args: Array<String>) {  
    println("Pi = $PI_From_Kotlin")  
    println("E = $E_From_Java")  
    println("sin(Pi) = ${Math.sin(PI_From_Kotlin)}")  
}
```

1.1.1. ПЕРЕМЕННЫЕ

Переменные только для чтения

Переменные только для чтения объявляются с использованием ключевого слова *val*.

Тип переменной *должен* быть задан явно в объявлении переменной если НЕ выполняется ее инициализация в месте объявления.

```
val a: Int
```

Тип переменной *может* быть задан явно в объявлении переменной если в объявлении выполняется ее инициализация:

```
val b: Int = 1
```

Присваивание переменной для чтения *необходимо* выполнить только один раз.

```
fun main() {  
    val x: Int = 1  
    val y: Int;  
    y = if (x > 0) 1 else -1  
}
```

Выведение типа переменной

Тип переменной может быть выведен из присваиваемого ей значения:

```
val c = 2
```

```
val PI = 3.14
```

```
val language = "Kotlin"
```

```
val d = if (c > 0) 1 else -1
```


Изменяемые переменные

Изменяемые переменные объявляются с использованием ключевого слова **var**.

```
var x: Int = 0  
var y = 0
```

```
fun increment() {  
    y = x  
}
```

Тип переменной не изменяется:

```
var size = 10
```

```
fun increment() {  
    size = "Kotlin" // Ошибка компиляции.  
}
```

Переменные (свойства) верхнего уровня

Переменные могут быть объявлены вне методов или классов (на верхнем уровне).

В таком случае они называются *свойства верхнего уровня*:

```
// Файл Colors.kt  
package diagram
```

```
val red    = "FF0000"  
val green  = "00FF00"  
val blue   = "0000FF"
```

В результате по имени файла будет создан класс *ColorsKt*, а свойства верхнего уровня будут *статическими переменными* этого класса.

```
import diagram.red  
  
var lineColor = red  
var textColor = diagram.green
```

1.1.1. ФУНКЦИИ

ФУНКЦИИ

- Функция начинается с ключевого слова *fun*
- Тип параметра указывается после имени параметра
- Тип результата вызова указывается после списка параметров. Этот тип отделен от списка параметров двоеточием.
- Тело функции может быть *блоком* (в фигурных скобках):

```
fun sum(a: Int, b: Int): Int {  
    return a + b  
}
```

- Тело функции может быть *выражением*:

```
fun sum(a: Int, b: Int): Int = a + b
```

- Условный оператор в языке Kotlin - это ТАКЖЕ выражение (вырабатывает значение):

```
fun maxOf(a: Int, b: Int): Int =  
    if (a > b) a else b
```

Вывод типа значения функции

- Если тело функции *выражение*, то тип возвращаемого значения может быть не указан. Этот тип выводится:

```
fun sum(a: Int, b: Int) = a + b
```

```
fun maxOf(a: Int, b: Int) = if (a > b) a else b
```

Функции не возвращающие значения

Если функция значение не возвращает, то тип возвращаемого значения опускается или это тип ***Unit***

```
var size = 10
```

```
fun increment() {  
    size++  
}
```

```
fun decrement(): Unit {  
    size++  
}
```

Тип ***Unit*** имеет единственное значение - объект ***Unit***. Это аналог типа ***void*** в Java.

Умалчиваемые аргументы функции

Для аргументов функций могут быть заданы умалчиваемые значения.

```
fun outMessage(text: String, kind: String = "Info") {  
    println("$kind: $text")  
}
```

```
outMessage("Start execution")
```

```
outMessage("Unknown identifier", "Error")
```

Таким образом можно избежать большого числа одноименных функций с различным количеством параметров.

Например, большого числа конструкторов.

Именованные аргументы функций

При вызове функции можно указывать имена параметров для которых задается значение.

```
fun getColor(red: Int = 0, green: Int = 0, blue: Int = 0) =  
    "rgb($red,$green,$blue)"
```

```
val darkBlue = getColor(blue = 100)
```

```
val lightRed = getColor(red = 255)
```

```
val violet = getColor(blue = 100, red = 200)
```

Таким образом можно избежать ошибок при неверном порядке значений однотипных аргументов функции.

При вызове из программы на языке Kotlin методов на языке *Java* именованные аргументы использовать нельзя.

В *Java* имена параметров в байткоде не хранятся.

Функции с переменным числом параметров

Модификатор ***vararg*** позволяет передавать в функцию переменное число параметров указанного типа.

```
fun printAll(vararg messages: String) {  
    for (m in messages) println(m)  
}
```

```
fun main() {  
    printAll("Hello", "Kotlin")  
}
```

Внутри функции с параметром ***message*** можно работать как с массивом ***Array***.

Функции с переменным числом параметров

(2)

Для дальнейшей передачи параметра *message* в другую функцию используется оператор `*`.

```
fun printAll(vararg messages: String) {  
    for (m in messages) println(m)  
}
```

```
fun log(vararg entries: String) {  
    printAll(*entries)  
}
```

```
fun main() {  
    printAll("Hello", "Kotlin")  
    log("Hello", "Kotlin")  
}
```

В этом случае он будет использоваться как *vararg messages: String* а не как *messages: Array*

Функции-расширения

При объявлении функции-расширения перед именем должен быть тип получателя. Этот тип расширяется.

```
// Файл StringsUtil.kt:  
package ch1.strings
```

```
fun String.lastChar(): Char = this.get(this.length - 1)
```

Ключевое слово **this** в функции расширения соответствует объекту-получателю. Его можно опустить:

```
fun String.lastChar(): Char = get(length - 1)
```

Импорт функций-расширений

Возможен импорт функций-расширений:

```
// Файл StringUtil.kt:
```

```
package strings
```

```
import strings.lastChar
```

```
val c = "Kotlin".lastChar()
```

Можно переименовать импортируемую функцию-расширение:

```
import strings.lastChar as last
```

```
val c = "Kotlin".last()
```

Использование функций-расширений из языка Java

Использование в программе на *Java* статического метода **getLastChar** из класса **StringUtilKt** расположенного в файле **StringUtil.kt**

```
/* Java */  
import strings.StringUtilKt;  
  
public class JavaMainFun {  
    public static void main(String[] args) {  
        StringUtilKt.lastChar("Kotlin");  
    }  
}
```

Свойства-расширения для чтения

// Файл Properties.kt

```
package strings
```

```
val String.lastChar: Char  
    get() = get(length - 1)
```

```
fun main() {  
    val kotlin = "kotlin"  
    println(kotlin.lastChar)  
}
```

Свойства-расширения для чтения и записи

// Файл Properties.kt

```
package string
```

```
var Array<Char>.firstChar: Char
```

```
    get() = get(0)
```

```
    set(value: Char) = this.set(0, value)
```

```
fun main() {
```

```
    var kotlinArray = arrayOf('k', 'o', 't', 'l', 'i', 'n' )
```

```
    kotlinArray.firstChar = 'K'
```

```
    kotlinArray.forEach { print(it) }
```

```
}
```

Использование свойств-расширений из языка Java

Использование в программе на *Java*
статического метода *lastChar* из класса *PropertiesKt*
расположенного в файле *Properties.kt*

```
/* Java */  
import strings.PropertiesKt;  
  
public class JavaMain {  
    public static void main(String[] args) {  
        PropertiesKt.getLastChar("Kotlin");  
  
        Character[] a = {'K', 'o', 't', 'l', 'i', 'n'};  
        PropertiesKt.setFirstChar(a, 'K');  
    }  
}
```


Функции-расширения - это статические функции

Функции расширения не модифицируют расширяемый класс. Это НЕ виртуальные функции расширяемого класса.
Переопределить функции-расширения нельзя.

```
open class Shape
```

```
class Rectangle: Shape()
```

```
fun Shape.getName() = "Shape"
```

```
fun Rectangle.getName() = "Rectangle"
```

```
fun printClassName(s: Shape) {  
    println(s.getName())  
}
```

```
printClassName(Rectangle())
```

Инфиксные функции

Функции помеченные ключевым словом **infix** могут вызываться в инфиксной нотации. Это значит, точка и скобки могут опускаться.

Дополнительные требования к инфиксным функциям:

- Должны быть членами-членами классов или функциями-расширениями.
- У них должен быть единственный параметр
- Параметр на принимает множественное число параметров
- Параметр не имеет умалчиваемого значения

```
var p1 = Point(1,1)
```

```
var p2 = Point(1,2)
```

```
var p3 = Point(2,1)
```

```
infix fun Point.line(p: Point): Point { ... }
```

```
infix fun Point.move(p: Point): Point { ... }
```

```
p1 line p2 move p3
```

```
p1.line(p2)
```

Функции-операторы

Обозначения операторов языка Kotlin могут быть использованы для создания новых функций-операторов.

Например, для сложения точек (экземпляров класса ***Point***) используется функция с модификатором ***operator***:

```
operator fun Point.plus(p: Point)  
    = Point(this.x + p.x, this.y + p.y)
```

```
var p1 = Point(1,1)
```

```
var p2 = Point(1,2)
```

```
fun main() {  
    println(p1 + p2)  
    println(p1.plus(p2))  
}
```

1.1.3. КЛАССЫ

Классы Java и Kotlin

Класс ***Point*** в языке ***Java***:

// Java

```
public class JavaPoint {  
    int x, y;  
  
    public JavaPoint(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

Класс ***Point*** в языке ***Kotlin***:

// Kotlin

```
class KotlinPoint(var x: Int, var y: Int)
```

Классы данных

```
data class Square(val v: Int, val h: Int)
```

Для класса данных автоматически генерируются методы:

- ***equals*** для сравнения экземпляров данных; данные равны если свойства равны.
- **hashCode** для использования экземпляров в качестве ключей в контейнерах на основе хэш-функций, таких как ***HashMap***
- ***toString*** для создания строкового представления, показывающего все поля в порядке их объявления

```
Square(v=1, h=4)
```

Классы-перечисления

Классы-перечисления используются для моделирования типов, которые представляют конечный набор различных значений.

```
enum class PieceKind {  
    PAWN, ROOK, KNIGHT, BISHOP, QUEEN, KING  
}  
fun main() {  
    val kind = PieceKind.PAWN  
    val message = when (kind) {  
        PieceKind.ROOK -> "ROOK"  
        PieceKind.KNIGHT -> "KNIGHT"  
        PieceKind.BISHOP -> "BISHOP"  
        PieceKind.PAWN -> "PAWN"  
        PieceKind.QUEEN -> "QUEEN"  
        PieceKind.KING -> "KING"  
    }  
    println(message)  
}
```

Свойства и методы классов-перечислений

```
enum class Color(val red: Int, val green: Int, val blue: Int) {  
    RED(255, 0, 0),  
    GREEN(0, 255, 0),  
    BLUE(0,0, 255),  
    VIOLET(255, 255, 0);  
  
    fun intensity() = red + green + blue  
}  
fun main() {  
    val red = Color.RED  
    println(red)  
    println(red.intensity())  
    println(Color.VIOLET.intensity())  
}
```


Запечатанные классы

- Запечатанные (*sealed*) классы ограничивают использование наследования.
- После объявления класса запечатанным, его потомки может быть могут быть объявлены только в том же файла, в котором объявлен запечатанный класс.

```
sealed class Figure(var x: Int, var y: Int)
```

```
class Point(x: Int, y: Int):
```

```
    Figure(x, y)
```

```
class Line(start: Point, var end: Point):
```

```
    Figure(start.x, start.y)
```

```
class Rectangle(start: Point, val end: Point):
```

```
    Figure(start.x, start.y)
```

```
class Circle(center: Point, var radius: Int):
```

```
    Figure(center.x, center.y)
```

Объекты “Одиночки” в языке Kotlin

Для одновременного объявления класса объекта и его единственного экземпляра (*Одиночка - Singleton*) используется ключевое слово **object**.

```
object DefaultListener : MouseAdapter() {  
    override fun mouseClicked(e: MouseEvent)  
    { ... }  
    override fun mouseEntered(e: MouseEvent)  
    { ... }  
}
```

DefaultListener.mouseClicked(e)

- К объекту обращаются по имени класса
- Объекты могут иметь предков и реализовывать интерфейсы
- Допускается вложенность объектов

Объекты-компаньоны

Объект-компаньон - это объект объявленный внутри класса. Синтаксически это похоже на статические поля и методы в Java.

```
class Diagram {  
    companion object Facture {  
        var defaultLineColor = Color.BLUE  
        var defaultTextColor = Color.BLUE  
        var defaultFillColor = Color.GREEN  
    }  
}  
fun main() {  
    Diagram.defaultLineColor = Color.RED  
}
```

- Определяется класс.
- Определяется объект-компаньон. Его название можно не указывать.
- Определяются поля и методы сопутствующего объекта.
- Используются поля и методы объект-компаньона через имя класса.

Объекты-выражения

- Объекты-выражения используются для создания анонимных объектов.
- Имя класса не требуется
- Имя объекта не требуется

```
var button = Button()
```

```
fun main() {  
    button.addMouseListener(  
        object: MouseAdapter() {  
            override fun mouseClicked(e : MouseEvent) =  
                println("event=$e")  
        }  
    )  
}
```

Анонимные объекты заменяют анонимные внутренние классы в Java.

Интерфейсы

```
public interface Move {  
    val piece: Piece?  
  
    @throws GameOver  
    fun doMove()  
  
    fun undoMove()  
}
```

1.1.3 УПРАВЛЯЮЩИЕ КОНСТРУКЦИИ

Циклы

```
class Board {  
    val squares = arrayListOf<Square>()  
}  
fun main() {  
    val board = Board();  
  
    for (square in board.squares) {  
        println("Board: $square")  
    }  
}
```

Оператор in

```
public interface Iterator<out T> {  
    public operator fun next(): T  
    ...  
}
```


Диапазоны (ranges)

- Диапазон представляет собой интервал между двумя значениями началом и концом.
- Диапазон включающий, второе значение всегда является частью диапазона.
- Диапазоны определяются с помощью *оператора ..* :

```
var oneToTen = 1 .. 10
```

Определение оператора в классе *Int*

```
class Int: Number() {  
    public operator fun rangeTo(other: Int): IntRange  
}
```

Объявление классов-диапазонов (ranges)

```
public open class IntProgression (  
    start: Int,  
    endInclusive: Int,  
    step: Int  
): Iterable<Int> { ... }
```

```
public class IntRange(start: Int,  
    endInclusive: Int)  
    : IntProgression(start, endInclusive, 1)  
    { ... }
```

Использование диапазонов в условном выражении

```
val x = 2
if (x in 1..5) {
    print("x is in range from 1 to 5")
}

if (x !in 6..10) {
    print("x is not in range from 6 to 10")
}
```

Использование числовых диапазонов в циклах

```
for (i in 0 .. 64) {  
    print( board.squares[i] )  
}
```

```
for(i in 0..7 step 2) {  
    print( board.squares[i] )  
}
```

```
for (i in 3 downTo 0) {  
    print( board.squares[i] )  
}
```

Инфиксная функции `downTo` и `step` в цикле

```
public infix fun Int.downTo(to: Int)
    : IntProgression {
    return IntProgression.fromClosedRange(this, to, -1)
}
```

```
public infix fun IntProgression.step(step: Int)
    : IntProgression
{
    return IntProgression.fromClosedRange
        (first, last, if (this.step > 0) step else -step)
}
```

Использование символьных диапазонов в циклах

```
for (c in 'a'..'d') {  
    print(c)  
}
```

```
for (c in 'z' downTo 's' step 2) {  
    print(c)  
}
```

Переопределение итератора

```
class Square(val v: Int, val h: Int)
```

```
class Board(val squares: List<Square>) {  
    operator fun iterator(): Iterator<Square> =  
        squares.iterator()  
}
```

```
fun main() {  
    for (square in board)  
        println("$square")  
}
```

Выбирающее предложение when

```
MyClass.cases(obj: Any) {  
  when (obj) {  
    1      -> println("One")  
    "Hello" -> println("Greeting")  
    is Long  -> println("Long")  
    !is String -> println("Not a string")  
    else     -> println("Unknown")  
  }  
}  
  
class MyClass {  
  fun main()  
    cases("Hello")  
}
```


Выбирающее выражение when

```
MyClass.whenAssign(obj: Any) =  
  when (obj) {  
    1      -> "One"  
    "Hello" -> "Greeting"  
    is Long -> "Long"  
    !is String -> "Not a string"  
    else    -> "Unknown"  
  }
```

```
class MyClass {  
  fun main()  
    println(whenAssign("Hello"))  
}
```

Условное выражение if

```
fun maxOf(a: Int, b: Int): Int =  
    if (a > b) a else b
```

Проверка равенства

```
val authors = setOf("Shakespeare", "Hemingway", "Twain")  
val writers = setOf("Twain", "Shakespeare", "Hemingway")
```

```
println(authors == writers)  
println(authors === writers)
```

Kotlin использует

- `==` для структурного сравнения

- `===` для сравнения ссылок

`**a == b**` компилируется в `**(a == null) b == null else a.equals(b)**`

- вернет **true** поскольку вызывает `authors.equals(writers)` а множество игнорирует порядок элементов
- вернет **false** поскольку различны ссылки `authors` и `writers`

1.1.4 ПЕРЕГРУЗКА УНАРНЫХ ОПЕРАТОРОВ.

Унарные префиксные операторы

Выражение	Транслируется в
+a	a.unaryPlus()
-a	a.unaryMinus()
!a	a.not()

Пример унарного префиксного оператора

```
data class Point(val x: Int, val y: Int)
```

```
operator fun Point.unaryMinus() = Point(-x, -y)
```

```
val point = Point(10, 20)  
println(-point)
```

1.1.5 ПЕРЕГРУЗКА БИНАРНЫХ ОПЕРАТОРОВ.

Арифметические операции

Выражение	Транслируется в
$a + b$	<code>a.plus(b)</code>
$a - b$	<code>a.minus(b)</code>
$a * b$	<code>a.times(b)</code>
a / b	<code>a.div(b)</code>
$a \% b$	<code>a.rem(b)</code>
<code>a..b</code>	<code>a.rangeTo(b)</code>


```
operator fun Point.plus(p: Point)
    = Point(this.x + p.x, this.y + p.y)
```

```
var p1 = Point(1,1)
```

```
var p2 = Point(1,2)
```

```
fun main() {
    println(p1 + p2)
    println(p1.plus(p2))
}
```

Оператор `in`

Выражение	Транслируется в
<code>a in b</code>	<code>b.contains(a)</code>
<code>a !in b</code>	<code>!b.contains(a)</code>

```
class Board(val nV: Int = 8, val nH: Int = 8) {  
var squares: Array<Array<Square>> =  
    Array(nV) { v ->  
        Array(nH) { h -> Square(v, h) } }  
}  
operator fun Board.contains(square: Square) =  
    (square.v in 0..nV-1) && (square.h in 0 until nH)  
class Piece (var square: Square)  
  
var piece: Piece = Piece()  
  
fun main() {  
    val chessBoard = ChessBoard(8, 8)  
    piece.square in chessBoard  
}
```

Оператор доступа по индексу

Выражение	Транслируется в
<code>a[i]</code>	<code>a.get(i)</code>
<code>a[i, j]</code>	<code>a.get(i, j)</code>
<code>a[i_1, ..., i_n]</code>	<code>a.get(i_1, ..., i_n)</code>
<code>a[i] = b</code>	<code>a.set(i, b)</code>
<code>a[i, j] = b</code>	<code>a.set(i, j, b)</code>
<code>a[i_1, ..., i_n] = b</code>	<code>a.set(i_1, ..., i_n, b)</code>

Квадратные скобки транслируются в вызов **get** или **set** с соответствующим числом аргументов.

```
class Board {
```

```
  private var squares: Array<Array<Square>> = ...
```

```
  operator fun get(v: Int, h: Int) = squares[v][h]
}
```

```
val board = Board(8, 8)
```

```
val s = board[2,3]
```

Оператор вызова

Оператор вызова (функции, метода) в круглых скобках транслируется в *invoke* с соответствующим числом аргументов.

Выражение	Транслируется в
a()	a.invoke()
a(i)	a.invoke(i)
a(i, j)	a.invoke(i, j)
a(i_1, ..., i_n)	a.invoke(i_1, ..., i_n)

Присвоения с накоплением

Выражение	Транслируется в
<code>a += b</code>	<code>a.plusAssign(b)</code>
<code>a -= b</code>	<code>a.minusAssign(b)</code>
<code>a *= b</code>	<code>a.timesAssign(b)</code>
<code>a /= b</code>	<code>a.divAssign(b)</code>
<code>a %= b</code>	<code>a.modAssign(b)</code>

Операторы равенства и неравенства

Выражение	Транслируется в
<code>a == b</code>	<code>a?.equals(b) ?: (b === null)</code>
<code>a != b</code>	<code>!(a?.equals(b) ?: (b === null))</code>

Операторы сравнений

Выражение	Транслируется в
<code>a > b</code>	<code>a.compareTo(b) > 0</code>
<code>a < b</code>	<code>a.compareTo(b) < 0</code>
<code>a >= b</code>	<code>a.compareTo(b) >= 0</code>
<code>a <= b</code>	<code>a.compareTo(b) <= 0</code>

Все сравнения транслируются в вызовы *compareTo*, от которых требуется возврат значения типа *Int*.

```
class Point(val x: Int, val y: Int) : Comparable<Point> {  
    private fun length() = x + y  
  
    override fun compareTo(p: Point): Int = when {  
        length() > p.length() -> 1  
        length() < p.length() -> -1  
        else -> 0  
    }  
}  
  
fun main() {  
    val b = Point(1, 1) < Point(2, 2)  
}
```

Пример операторов + для Square

```
class Board {  
    private var squares: Array<Array<Square>>  
    operator fun get(v: Int, h: Int)  
        = squares[v][h]  
}  
class Square(val board: Board,  
             val v: Int,  
             val h: Int) {  
    var piece: Piece? = null  
}  
operator fun Square.plus(step: Int): Square?  
    = this.board[this.v, this.h + step]  
operator fun Square.minus(step: Int): Square?  
    = board[v, h - step]  
operator fun Square.plus(dir: Dirs): Square?  
    = board[v + dir.dv, h + dir.dh]
```

Пример оператора + для Piece

abstract

```
class Piece(var square: Square,  
            var color: PieceColor)
```

```
operator fun Piece.plus(step: Int): Square?  
    = square + step
```

```
operator fun Piece.minus(step: Int): Square?  
    = square - step
```

```
operator fun Piece.plus(dir: Dirs): Square?  
    = square + dir
```

Использование оператора + для Piece

```
val Piece.next: Square?  
get() = when (this.color) {  
    PieceColor.WHITE -> this - 1  
    PieceColor.BLACK -> this + 1  
    PieceColor.GREEN -> this + Dirs.LEFT  
    PieceColor.BLUE -> this + Dirs.RIGHT  
}
```

1.1.3 ИСКЛЮЧЕНИЯ

Порождение исключения

```
@Throws(GameOver::class)
override fun doMove() {
    //...
    int enemies = piece.getEnemies().size();
    int friends = piece.getFriends().size();

    if (enemies == friends)
        throw new GameOver(GameResult.DRAWN);
}
```


Перехват исключения

```
fun startGame() {  
    while (true) {  
        val player = players[moveColor]  
        if (player === IPlayer.HOMO_SAPIENCE)  
            break // Ход сделает человек.  
  
        try { player!!.doMove(this, moveColor) }  
        catch (e: GameOver) { break }  
        moveColor = getOpponentColor(moveColor)  
    }  
}
```

Исключения это выражения

```
val number = try {  
    Integer.parseInt( reader.readLine( ) )  
} catch (e : NumberFormatException) {  
    null  
}
```