

Kotlin – pokračovanie

Cheat sheets

- <https://www.programming-idioms.org/cheatsheet/Kotlin>
- <https://github.com/vmandro/Prednasky/tree/master/Kotlin>

The billion-dollar mistake

I call it my billion-dollar mistake. It was the invention of the **null** reference in 1965...This has led to innumerable errors, vulnerabilities, and system crashes, which have probably caused a billion dollars of pain and damage in the last forty years.

Kotlin Null Safety

Sir Tony Hoare

FRS FREng



Tony Hoare in 2011

Born	Charles Antony Richard Hoare 11 January 1934 (age 85) Colombo, British Ceylon
Residence	Cambridge
Other names	C. A. R. Hoare
Alma mater	University of Oxford (BA) Moscow State University
Known for	Quicksort Quickselect Hoare logic Null reference Communicating Sequential Processes Structured programming
Awards	Turing Award (1980) Harry H. Goode Memorial Award (1981) Faraday Medal (1985) Computer Pioneer Award (1990) Kyoto Prize (2000) IEEE John von Neumann Medal (2011)

Nullables

To, čo je

- Optional v Java, resp.
- Option v Scala, resp. kde inde

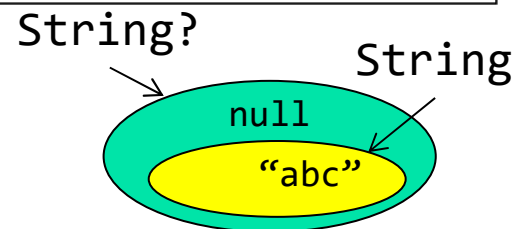
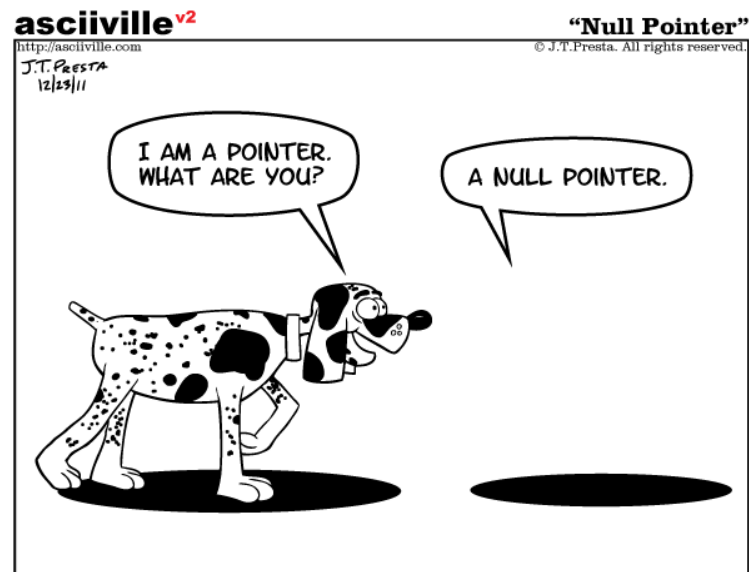
Napr. `String?` je typ pre reťazec alebo null

Ale `String` je typ len pre SKUTOČNÝ REĹAZEC, not-null

Preto `a:String?` nemôžete priradiť do `b:String`, lebo čo, ak by `a == null`

Ak ste skalo-pevne presvedčený, že hodnota `a:String? != null`, môžete opatrne použiť BANG-BANG (`!!`) operátor a oklamať type-checker
`val b:String = a!!`

Ak ale neviete, či `a:String? == null`, tak použijete tzv. **Elvis operátor**
`val c:String = a?: "default, ak je prázdny reťazec"`






Nullables

(ďalšie operátory na konverziu medzi type a type?)



- 
- Elvis operátor
`obj?:default` = if (obj == null) default else obj
 - Safe call operátor (Elvis na Žižku)
`obj?.m()` = if (obj == null) null else obj.m()
 - Not-null assertion (bang-bang !!)
`obj!!` = if (obj != null) obj else N.P.E. – null pointer Ex.
 - Safe cast
`obj as? T` = if (obj typeof T) obj else null
`obj as T` = if (!obj typeof T) cast exception
 - let
`obj?.let {...it...}` = if (obj != null) {...it <- obj...}

Nullables

(ešte raz, podrobnejšie)



V Jave je typ String skutočný reťazec alebo null

V Kotlině String je **LEN skutočný reťazec** a null nepatrí do typu String

Existuje String? čo je String alebo null, vo všeobecnosti: $T? = T \cup \text{null}$

T? Podobne vo Swingu, Java Optional[T] =, Scala Option[T]

```
fun foo(str : String?) {  
    println(str)  
    if (str != null) println(str.toUpperCase())  
    println(str?.toUpperCase()) // safe call operátor  
                                // x?.m == if (x != null) x.m else null  
}  
  
fun stringLen(s: String?): Int = s?.length?:0 // Elvis operátor  
if (if (s == null) then null else s.length) == null then 0 else s.length  
  
fun nonEmptystringLen(s: String?): Int {  
    val sNotNull: String = s!! // určite nebude null,  
    // ak bude tak exception kotlin.KotlinNullPointerException  
    return sNotNull.length  
}
```



Properties

```
data class Rectangle(val height: Int, val width: Int) {  
    val isSquare: Boolean  
        get() { return height == width }  
    val area: Int  
        get() { return height * width }  
    fun size(): Int { return height * width }  
    fun size_(): Int = height * width  
}
```

```
fun main(args: Array<String>) {  
    val rect = Rectangle(41, 43)  
    println("Toto $rect je stvorec: ${rect.isSquare}")  
    println("Obsah $rect je: ${rect.size()}")  
    println("Area $rect je: ${rect.area}")  
}
```



Enumerables, when

(sú aj v Jave 5+)

```
enum class Color { RED, ORANGE, YELLOW, GREEN, BLUE, INDIGO, VIOLET }
```

```
enum class Colour(val r: Int, val g: Int, val b: Int) {  
    WHITE(0, 0, 0), RED(255, 0, 0), YELLOW(255, 255, 0),  
    GREEN(0, 255, 0), BLUE(0, 0, 255), BLACK(255, 255, 255);  
    fun rgb() = (r * 256 + g) * 256 + b  
}
```

```
fun getMnemonic(c : Colour) = {  
    when (c) {  
        Colour.WHITE -> "Biela"  
        Colour.BLACK -> "Biela"  
        else          -> "Seda"  
    }  
}
```



Enumerables, when

(when alias switch)

```
fun mix(c1: Color, c2: Color) =  
    when (setOf(c1, c2)) {  
        setOf(Color.RED, Color.YELLOW) -> Color.ORANGE  
        setOf(Color.YELLOW, Color.BLUE) -> Color.GREEN  
        setOf(Color.BLUE, Color.VIOLET) -> Color.INDIGO  
        else -> throw Exception("Dirty color")  
    }
```

```
fun mixOptimized(c1: Color, c2: Color) =  
    when {  
        (c1 == Color.RED && c2 == Color.YELLOW) -> Color.ORANGE  
        (c1 == Color.YELLOW && c2 == Color.BLUE) -> Color.GREEN  
        (c1 == Color.BLUE && c2 == Color.VIOLET) -> Color.INDIGO  
        else -> throw Exception("Dirty color")  
    }
```



Derivácia

```
interface Expr // abstract class
enum class Operator { Plus, Times }
data class Num(val value: Int) : Expr // subclass, extends
data class Variable(val variable: String) : Expr
data class Op(val operator: Operator, val left: Expr, val right: Expr): Expr

fun derive(e: Expr, variable : String): Expr {
    if (e is Num) { return Num(0) } // typeof
    else if (e is Variable) { // typeof
        return if (e.variable == variable) Num(1) else Num(0) // typecast
    } else if (e is Op) {
        when(e.operator) { // vie, že e:Expr je Op, ((Op)e).operator
            Plus -> return Op(Plus,
                derive(e.left, variable), derive(e.right, variable))
            Times -> return Op(Plus,
                Op(Times, derive(e.left, variable), e.right),
                Op(Times, derive(e.right, variable), e.left))
        }
    }
    throw IllegalArgumentException("Unknown expression")
}
```


Zjednodušovanie

```
fun simplify(e: Expr):Expr {  
    when (e) {  
        is Op -> {  
            when (e.operator) {  
                Operator.Plus -> {  
                    return  
                    if (e.left is Num && e.right is Num)  
                        Num(e.left.value + e.right.value)  
                    else if (e.left == Num(0)) simplify(e.right)  
                    else if (e.right == Num(0)) simplify(e.left)  
                    else e.copy(left = simplify(e.left), right = simplify(e.right))  
                }  
            }  
        }  
    }  
    return e  
}
```



Metódy

```
sealed class Expression {  
    data class Num(val value: Int) : Expression()  
    data class Variable(val variable: String) : Expression()  
    data class Op(val operator: Operator,  
        val left: Expression, val right: Expression) : Expression()
```

```
fun derive(variable : String): Expression {  
    if (this is Num) { // typeof  
        return Num(0)  
    } else if (this is Variable) {  
        return if (this.variable == variable) Num(1) else Num(0)
```

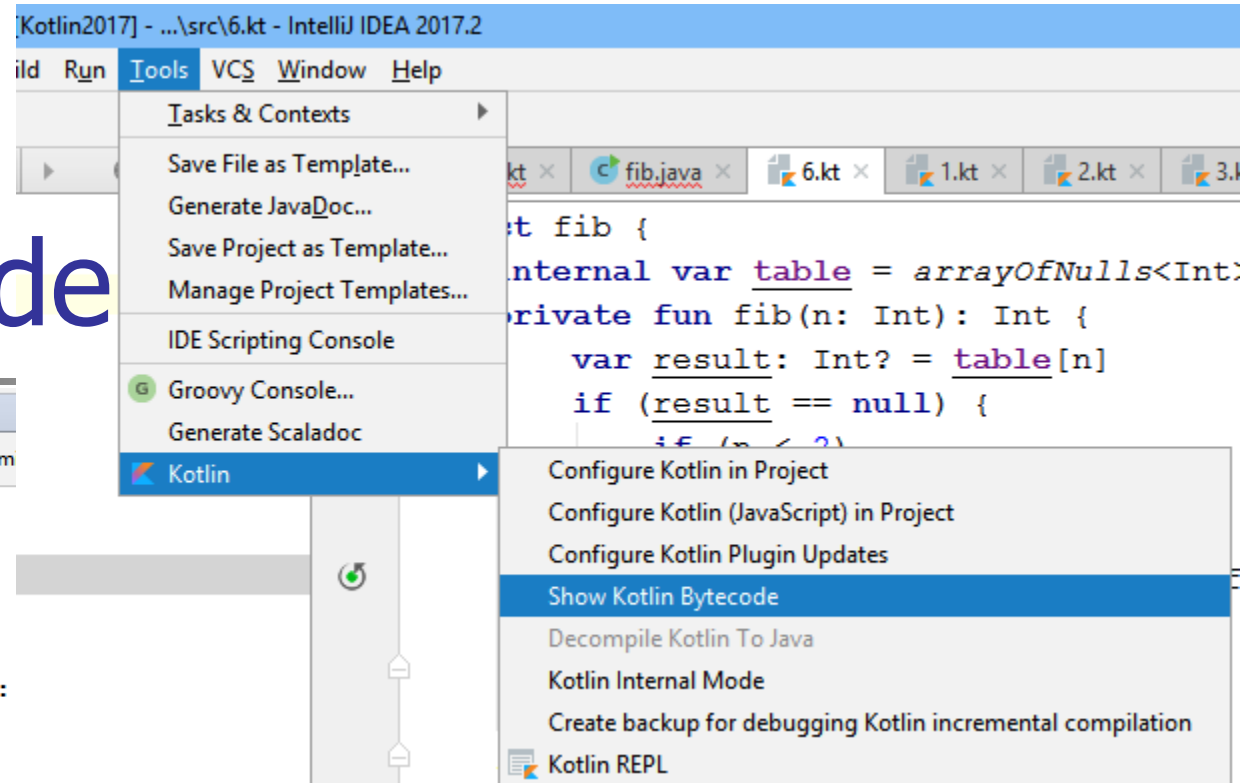
```
fun simplify(): Expression {  
    when (this) {  
        is Op -> {
```

ByteCode

Kotlin Bytecode

☒ Decompile ☒ Inline ☒ Optimize

```
// access flags 0x12
private final fib(I)I
L0
    LINENUMBER 4 L0
    GETSTATIC fib.table :
    ILOAD 1
    AALOAD
    ASTORE 2
L1
    LINENUMBER 5 L1
    ALOAD 2
    IFNONNULL L2
L3
    LINENUMBER 6 L3
    ILOAD 1
    ICONST_2
    IF_ICMPGE L4
L5
    LINENUMBER 7 L5
    ICONST_1
    INVOKESTATIC java/lang/Integer.valueOf (I)Ljava/lang/
    ASTORE 2
    GOTO L6
L4
    LINENUMBER 9 L4
```



Videli ste už niekedy kompilát vášho kódu

Decompile

(vidíte len časť main – ale je to .java)



@JvmStatic

```
public static final void main(@NotNull String[] args) {
    Intrinsics.checkNotNull(args, "args");
    int i = 0;
    byte var2 = 19;
    if (i <= var2) {
        while(true) {
            String var3 = "fib(" + i + ")=" + INSTANCE.fib(i);
            System.out.println(var3);
            if (i == var2) {
                break;
            }
            ++i;
        }
    }
}
```

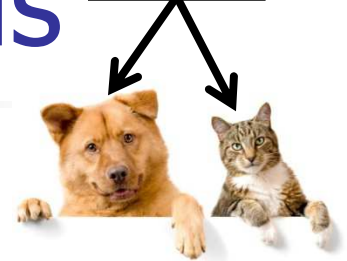


Immutableables

Collection	Immutable	Mutable
List	<code>listOf()</code> <i><code>listOf<String>("a", "b")</code> <code>.get(0)</code></i>	<code>arrayListOf()</code> <i><code>arrayListOf<String>("a", "b")</code> <code>.set(1, "Kotlin")</code></i>
Set	<code>setOf()</code> <i><code>setOf<String>("a", "b", "a")</code> <code>.contains("a")</code></i>	<code>hashSetOf()</code> <code>linkedSetOf()</code> <code>sortedSetOf()</code> <i><code>hashSetOf<String>("a", "b", "a")</code> <code>.remove("a")</code></i>
Map	<code>mapOf()</code> <i><code>mapOf("a" to 1, "b" to 100)</code> <code>.keys</code></i>	<code>hashMapOf()</code> <code>linkedMapOf()</code> <code>sortedMapOf()</code> <i><code>hashMapOf("a" to 1, "b" to 100)</code> <code>.set("b", 10)</code></i>

Podtriedy a polymorfizmus

Zviera



```
open class Zviera {                                // open znamená nie final
    open fun pozdrav() { }                          // open znamená nie final
}
class Macka : Zviera() {                          // Macka je podtrieda Zviera
    override fun pozdrav() { println("mnau") }
}
class Pes : Zviera() {                            // Pes je ine Zviera
    override fun pozdrav() { println("haf") }
}

class Stado<T : Zviera>() {                        // stádo implementujeme ako
    var lst: MutableList<T> = mutableListOf()      // mutable list je zámer
    val size: Int get() = lst.size
    operator fun get(i: Int): T {                  // T je v out pozíci
        return lst.get(i); }                      // operátor dovolí stado[i]
    operator fun set(i: Int, v: T) {              // T je v in pozíci
        lst.set(i, v)                             // operátor dovolí stado[i] = v
    }
}
```

Podtriedy a polymorfizmus

(variance – covariancia a contravariancia – teória)



Macka je podtrieda Zviera, Macka <: Zviera

Pes je podtrieda Zviera, Pes <: Zviera

Stado<T : Zviera> je parametrický typ pre ľubovoľný podtyp T typu Zviera

Stado<Macka> ani Stado<Pes> ale nie je podtrieda Stado<Zviera>

Stado je na parameter T **invariantné**

Ak ale chceme, aby Stado<Macka>, Stado<Pes> BOLI podtrieda Stado<Zviera>, horoví sa tomu **covariancia**, potom stado musí byť deklarované takto:

```
class Stado<out T : Zviera>() { ... } // Stado[Macka] <: Stado[Zviera]
```

Ak chceme, aby XYZ<Zviera> BOLA podtrieda XYZ<Macka>, horoví sa tomu **contravariancia**, potom stado musí byť deklarované takto:

```
class XYZ<in T : Zviera>() { ... } // XYZ[Zviera] <: XYZ[Macka]
```

Podtriedy a polymorfizmus

(stado je invariantné)

```
fun pozdravitVsetky(zvery : Stado<Zviera>) {  
    for (i in 0 until zvery.size)  
        zvery[i].pozdrav()  
}
```

```
fun pozdravitMacky(macky : Stado<Macka>) {  
    for (i in 0 until macky.size)  
        macky[i].pozdrav()           // macky[i] : Macka, preto .pozdrav()
```

```
    pozdravitVsetky(macky)           // toto nejde lebo macky : Stado<Macka>  
                                     // to nie je podtyp Stado<Zviera>
```

```
    pozdravitVsetky(macky as Stado<Zviera>) // smart Cast  
        // povie kompilátoru, že ver mi, macky : Stado<Zviera>  
        // kompilátor uverí a zavolá funkciu
```

```
    pozdravitVsetky(macky)           // toto už ide, lebo kompilátor uveril  
                                     // že macky : Stado<Zviera>
```

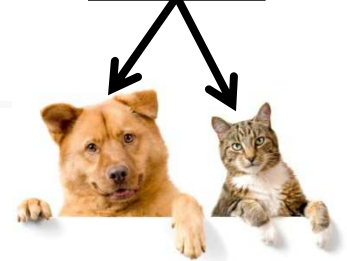
```
}
```



Podtriedy a polymorfizmus

(stado je invariantné a zneužijeme toho)

Zviera



```
val stado = Stado<Macka>()           // main
stado.append(Macka())
stado.append(Macka())
stado[1] = Macka()                   // ilustrácia operátora set
val m = stado[0]                     // ilustrácia operátora get
pozdravitMacky(stado)
```

```
stado[1] = Pes()                     // nejde, lebo Macka nie je podtrieda Pes
stado.append(Pes())                  // nejde, lebo Macka nie je podtrieda Pes
pozdravitVsetky(stado)               // Stado<Macka> nie je podtrieda Stado<zviera>
                                     // tzv. Smart cast
pozdravitVsetky(stado as Stado<Zviera>) // ale presvedčíme kompilátor
stado[1] = Pes()                     // a už nám verí
stado.append(Pes())                  // oklamali sme ho 😊 😊 😊 😊 😊
```

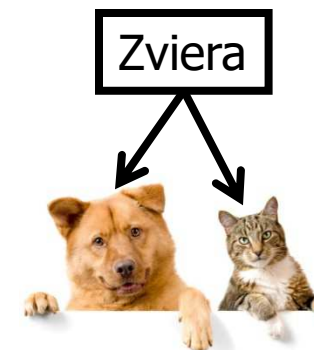
```
pozdravitVsetky(stado)               // stado as Stado<Zviera> to on už vie !
pozdravitMacky(stado)                // toto on kompilátor vie, ale keďže
                                     // sme ho oklamali, vypomstí sa nám v runtime
```

Exception "main" java.lang.ClassCastException:Pes cannot be cast to Macka
Hádanka: na ktorom riadku to padlo ?

13.kt

Covariancia

(prvý pokus - stado snád' bude covariantné)



Ak ale chceme, aby Stado<Macka>, Stado<Pes> BOLI podtriedy Stado<Zviera>, tak to **nejde** takto:

```
class Stado<out T : Zviera>() {           // v scale Stado[+T]
    var lst: MutableList<I> = mutableListOf()
        // T je deklarované ako out je v invariant pozícií
    val size: Int get() = lst.size
    operator fun get(i: Int): T { return lst.get(i); }
        // T je deklarované ako out je v out pozícií, ok 😊
    operator fun set(i: Int, v: I) { lst.set(i, v) }
        // T je deklarované ako out je v in pozícií
    fun append(v: I) { lst.add(v) }
        // T je deklarované ako out je v int pozícií
}
Scala: covariant argument in contravariant position ...
```

Veľmi zjednodušene:

out je výstupný argument, **in** je vstupný argument metódy

Viac: <https://kotlinlang.org/docs/reference/generics.html>

Covariancia

(druhý pokus - stado už bude covariantné za cenu ...)



Ak ale chceme, aby `Stado<Macka>`, `Stado<Pes>` BOLI podtriedy `Stado<Zviera>`:

- nesmie mať žiadnu metódu so vstupným argumentom `:T`, lebo ten je out
- ako štruktúru naplniť, modifikovať ? jedine v konštruktore
- ergo, je to nemodifikovateľná **[immutable]** štruktúra/trieda/typ

```
class Stado<out T : Zviera>(val lst : List<T>) {  
    val size: Int get() = lst.size  
    operator fun get(i: Int): T { return lst.get(i); }  
    // T je deklarované ako out je v out pozícií, ok 😊  
    //operator fun set(i: Int, v: T) { lst.set(i, v) }  
    //fun append(v: T) { lst.add(v) }  
    // var lst2: MutableList<T> = mutableListOf()  
}  
  
fun pozdravitMacky(macky : Stado<Macka>) {  
    pozdravitVsetky(macky)           // toto ide lebo macky:Stado<Macka>,  
}                                     // to je podtyp Stado<Zviera>  
  
val stado = Stado<Macka>(ListOf(Macka(), Macka()))  
pozdravitVsetky(stado)
```

Contravariância

()



```
abstract class Zviera(val size : Int = 0) { }  
data class Macka(val krasa : Int) : Zviera(1) { }  
data class Pes(val dravost : Int) : Zviera(2) { }
```

// alias comparable

```
interface Compare<in T> {  
    fun compare(z1: T, z2: T): Int  
}
```

Contravariância (in):
Macka <: Zviera =>
Compare[Zviera] <: Compare[Macka]

```
val MackaCompare : Compare<Macka> = object: Compare<Macka> {  
    override fun compare(m1: Macka, m2: Macka): Int {  
        println("macky$m1 a $m2 si porovnavaju ${m1.krasa} a ${m2.krasa}")  
        return m1.krasa - m2.krasa  
    }  
}
```

podtyp

nadtyp

// val ZvieraCompare: Compare<Zviera> = MackaCompare // pre contravar...

```
val ZvieraCompare: Compare<Zviera> = object: Compare<Zviera> {  
    override fun compare(z1: Zviera, z2: Zviera): Int {  
        println("zviera $z1 a $z2 si porovnavaju ${z1.size} a ${z2.size}")  
        return z1.size - z2.size  
    }  
}
```



Zhrnutie

(covariancia, contravariancia, invariancia)

Covariant	Contravariant	Invariant
Producer<out T>	Consumer<in T>	MutableList<T>
$T_1 <: T_2 \Rightarrow G[T_1] <: G[T_2]$ Príklad: Producer<Macka> je podtyp Producer<Zviera> Skutočný príklad: interface List<out E>: Collection<E>	$T_1 <: T_2 \Rightarrow G[T_2] <: G[T_1]$ Príklad: Consumer<Zviera> je podtyp Consumer<Macka> Skutočný príklad: Interface Comparable<in E>	$T_1 <: T_2 \Rightarrow$ G[T ₁] a G[T ₂] nemajú ŽIADEN vzťah
T môže byť len v out pozícií, napr. výsledok fcie	T môže byť len v in pozícií, napr. vstup do fcie	T môže byť v ľubovoľnej pozícií
https://kotlinlang.org/docs/reference/generics.html		



No fajn, hneď je to jasnejšie 😊



Kotlin: má pre co/contra-variáciu out/in



Scala: +/-

A ako to bolo v Jave ?

- Stado<? extends Zviera>
- Compare<? Super Macka>