Zaawansowane programowanie obiektowe

Lecture 7
(Kotlin)

Szymon Grabowski, Wojciech Bieniecki
sgrabow@kis.p.lodz.pl , wbieniec@kis.p.lodz.pl
http://szgrabowski.kis.p.lodz.pl/zpo18/

Łódź, 2018

Kotlin – założenia języka

- „lepsza Java”
- interoperacyjność z językami działającymi na JVM
- język do zastosowań profesjonalnych / produkcyjnych (pragmatyzm, a nie akademickość)
- szybka kompilacja
- do desktopów, serwerów i aplikacji mobilnych (Android)
- brak typów prymitywnych
- eliminacja błędów odwołania (dereferencji) (ang. *null-pointer safety*)
- brak ambicji. ☺ W przeciwieństwie do Scali,
- Kotlin nie ma np. osobnych kolekcji (wykorzystuje JFC)

Krótka historia

• początki: JetBrains, 2011—2012 (otwarty kod od lutego 2012)

•natywna obsługa w IntelliJ IDEA 15+

•Google I/O 2017: Kotlin staje się oficjalnym językiem pod platformę Android

•obsługa Kotlina w Android Studio 3.0 (paźdz. 2017)

•aktualna wersja 1.3.10 (list. 2018).

•Od wersji 1.3 można pisać

fun main() …

zamiast

fun main(args: Array<String>) …

Drobne ulepszenia i zmiany

opcjonalne średniki

typ po nazwie zmiennej/parametru (jak w Scali), tj.

```
a: Int
```

zamiast

```
int a
```

var / val – jak w Scali

brak wyjątków kontrolowanych (tj. wszystkie unchecked)

tworzenie obiektów bez słowa kluczowego new

```
val n = BigInteger(12345678)
```

przypisania są wyrażeniami (ang. expressions), nie instrukcjami (ang. statements), np.:

```
while ((rem = a % b) != 0) {
   a = b
   b = rem
}
```

Inferencja typów

Inferencja typu polega na tym, że kompilator sam oblicza typ na podstawie wyrażenia, którym obiekt jest inicjalizowany.

```
fun double(x: Int) = x * 2
```

zamiast

```
fun double(x: Int): Int = ...)
```

```
val s = "abc"
```

zamiast

```
val s: String = "abc"
```

Praca z Kotlinem

Z linii poleceń: pobrać Kotlin Compiler.

hello.kt

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

Program można kompilować do kodu klasy i uruchamiać:

```
d:\>kotlinc hello.kt

d:\>kotlin HelloKt
Hello, World!
```

Można tworzyć archiwum z JavaRuntime i uruchamiać Javą:

```
d:\>kotlinc hello.kt -include-runtime -d hello.jar

d:\>java -jar hello.jar
Hello, World!
```

Praca z Kotlinem

Można tworzyć skrypt i uruchamiać bez kompilacji:

hello.kts

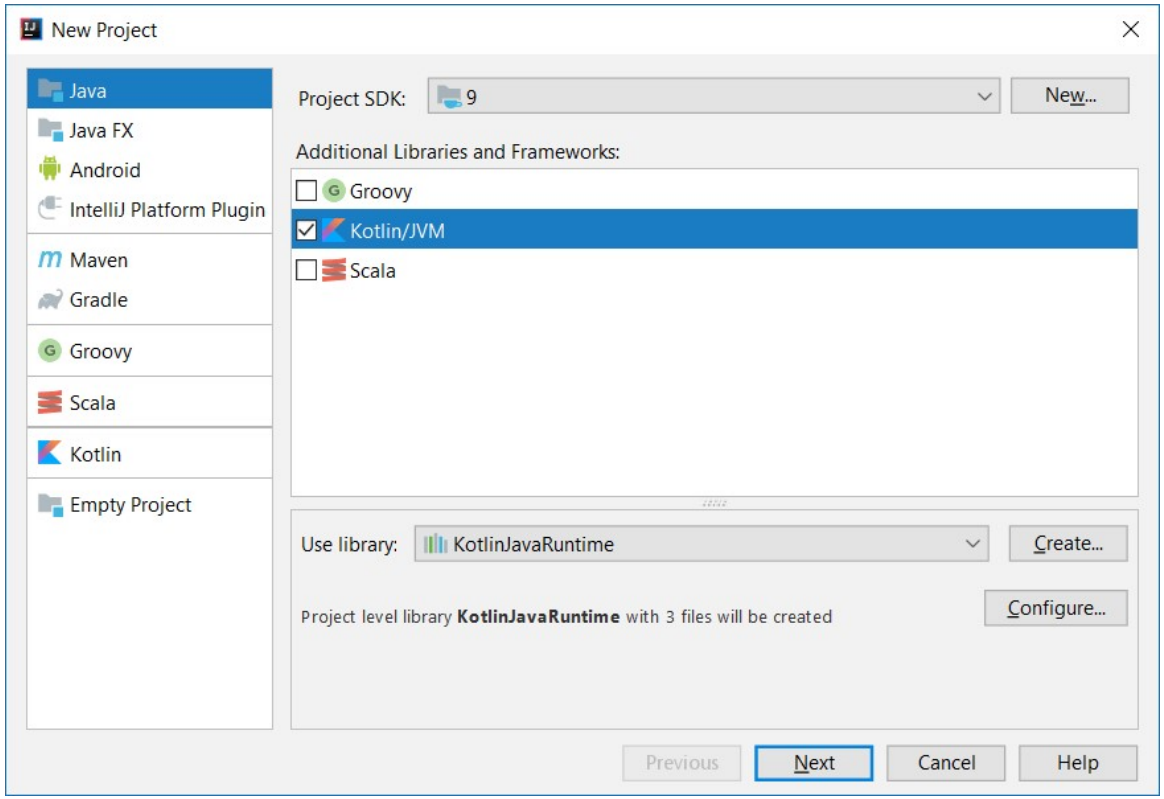
```
println("Hello, World!")
```

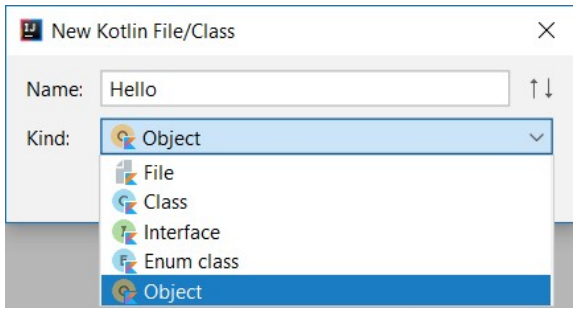
```
d:\>kotlinc -script hello.kts
Hello, World!
```

Można uruchomić konsolę REPL:

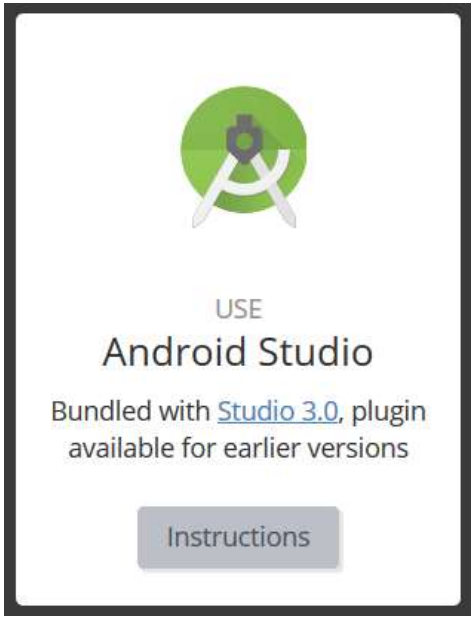
```
d:\>kotlinc
Welcome to Kotlin version 1.3.10 (JRE 1.8.0_144-_2017_08_24_19_19-b00)
Type :help for help, :quit for quit
>>> :load hello.kts
Hello, World!
>>> println("Witaj Kotlinie!")
println("Witaj Kotlinie!")Witaj Kotlinie!
>>> 2+3
5
>>>
```

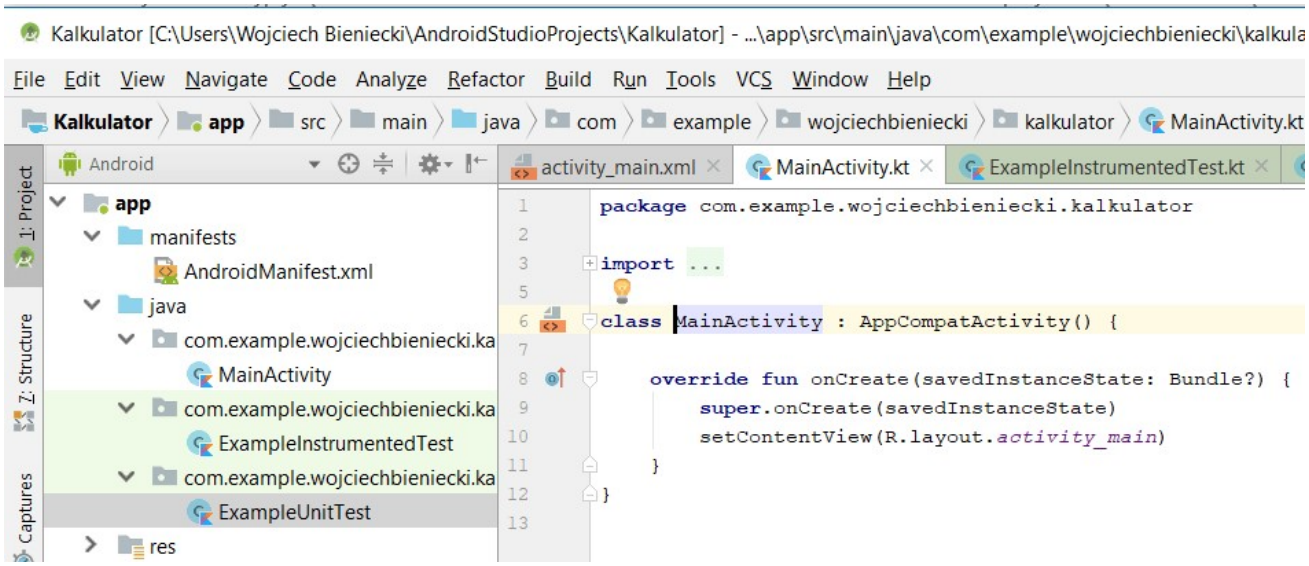
Praca z Kotlinem





Praca z Kotlinem





Android Studio wspiera konwersję kodu z Javy na Kotlin

Typy liczbowe

Double, Float, Long, Int, Short, Byte
odpowiedniki typów prymitywnych Javy.

Char NIE jest typem liczbowym, ale podlega konwersji.

BRAK niejawnej konwersji poszerzającej (Int --> Float):

```
var a = 10L  // a is a Long literal, note the L
var b = 20
var a = b  // this won't work
var a = b.toLong()  // this will work
```

Funkcje konwersji: to[Byte|Short|Int|Long|Float|Double|Char].

Każdy obiekt typu liczbowego może być przekonwertowany do innego!
Mogą wyjść bzdury:

```
val a: Byte = 127  // -128..127 are OK
var b: Int = 1233
val c = b.toByte()
println("$a $b $c")  // 127 1233 -74
```

Typy liczbowe całkowite bez znaku

(Kotlin 1.3+, w tej chwili funkcjonalność eksperymentalna)

Definiowanie typów bez znaku z użyciem literałów

```
val uint = 42u
val ulong = 42uL
val ubyte: UByte = 255u
```

Konwersja pomiędzy typami ze znakiem i bez znaku

```
val int = uint.toInt()
val byte = ubyte.toByte()
val ulong2 = byte.toULong()
```

Operatory analogiczne do typów ze znakiem

```
val x = 20u + 22u
val y = 1u shl 8
val z = "128".toUByte()
val range = 1u..5u
```

Obiektowość w Kotlinie

Klasy (oprócz abstrakcyjnych) są domyślnie finalne.

Słowa kluczowe: open, override. Dwukropek zamiast extends.

```
// open on the class means this class will allow derived classes
open class MegaButton  {

    // no-open on a function means that
    //    polymorphic behavior disabled if function overridden in derived class
    fun disable() { ... }

    // open on a function means that
    //    polymorphic behavior allowed if function is overridden in derived class
    open fun animate() { ... }
}

class GigaButton: MegaButton {

    // Explicit use of override keyword required to override a function in derived class
    override fun animate() { println("Giga Click!") }
}
```

Klasy są też domyślnie publiczne.

Obiektowość w Kotlinie, c.d.

Klasy są też domyślnie publiczne.

Modyfikatory widoczności (słowa kluczowe):
public, internal, protected, private.

When applied to a class member:

```
public (default): Visible everywhere
internal:         Visible in a module
protected:        Visible in subclasses
private:          Visible in a class
```

When applied to a top-level declaration

```
public (default):  Visible everywhere
internal:          Visible in a module
private:           Visible in a file
```

Konstruktory

Znów inspiracja Scalą:
konstruktor główny (ang. *primary constructor*)
i pomocnicze (*secondary constructors*).

```
class BankAccount {
    var accountBalance: Double = 0.0
    var accountNumber: Int = 0

    constructor(number: Int, balance: Double) {
        accountNumber =  number
        accountBalance = balance
    }

    fun displayBalance()
    {
       println("Number $accountNumber")
       println("Current balance is $accountBalance")
    }
}
```

Operatory

zmiany w stosunku do Javy

Brak typów prymitywnych w Kotlinie ma konsekwencje…

W porównaniach nie używamy equals()
(choć wolno to zrobić), tylko == oraz !=.
Te operatory wewnętrznie wołają metodę equals.

Dla porównania adresów (ang. *referential equality*),
używamy === i !==.

<, >, <=, >= tłumaczone na odwołania do compareTo().

Listy/mapy: dodano odwołanie przez […] (nie tylko get).

Operatory || i && jak w Javie, ale brak LOGICZNYCH operatorów | i &.
Zamiast nich są słowa or, and.

Triki a'la Python lub Scala

Wypakowanie argumentów z użyciem * (*spread operator*)

```
fun main(args: Array<String>) {
  val list = listOf("args: ", *args)
  println(list)
}
```

Możliwe również użycie * w środku listy argumentów, w stylu:
fun printNumbers(vararg numbers: Int) { … }
printNumbers(10, *numbers, 30, 40)

Dekonstrukcja (obiekt wypakowany do krotki)

```
for ((index, element) in collection.withIndex()) {
  println("$index: $element")
}
```

Interpolacja napisów (*string templates*)

Przykład – patrz wyżej. Albo np.

```
val s = "abc"; println("Length of s1 string is ${s1.length}.")
```

Lambda

```
val allowedUsers = users.filter { it.age > MINIMUM_AGE }
```

Wyrażenie lambda ma często tylko jeden parametr.
Jeśli kompilator jest w stanie odgadnąć sygnaturę,
to można pominąć deklarację parametru i znak ->.
Parametr niejawnie zadeklarowany ma wtedy nazwę it.

Ale taki zapis też jest poprawny:

```
val allowedUsers = users.filter { x -> x.age > MINIMUM_AGE }
```

A także: ... = users.filter({ ...})

Wartość zwracana z lambdy to wartość ostatniego wyrażenia, np.:

```
val evenPositive = ints.filter {
    val positive = it > 0
    positive and (it % 2 == 0)
}
```

Przykład operacji na kolekcjach

Zadanie: mamy listę osób, chcemy je pogrupować wg wieku.

```
val persons = listOf(Person("Max", 18), Person("David", 12),
            Person("Peter", 23), Person("Pamela", 23))
```

→ mapa: {18=Max, 23=Peter;Pamela, 12=David}

```
// w Javie:
Map<Integer, String> map = persons
   .stream()
      .collect(Collectors.toMap(
         p -> p.age,
         p -> p.name,
         (name1, name2) -> name1 + ";" + name2));

// w Kotlinie:
val map = persons.groupBy { it.age }.
   mapValues { it.value.joinToString(";") { it.name } }
```

Pętle

Oprócz for, while i do-while jest też:

- foreach, np.:

```
someIterable.forEach {
    print(it.toString())
}
```

- repeat, np.:

```
repeat(10) { i -> println("Jesteśmy w ${i+1}-wszej linii.") }
```

break i continue mogą odwoływać się do etykiet, np.:

```
outer@ for(i in 0..10) {
   inner@ for(j in 0..10) {
      break      // wychodzi z pętli WEWN.
      break@inner  // wychodzi z pętli WEWN.
      break@outer  // wychodzi z pętli ZEWN.
   }
}
```

Tablice

Typ Array. Można tworzyć przez funkcję arrayOf.

```
val arr = arrayOf(1, 2, 5)
val squares = Array(10, { i -> i * i })
```

W przeciwieństwie do Javy,
tablica nie jest elementem języka, ale klasą kolekcji.
Dostępne funkcje (m.in.) size, get, set.

Ale get/set dostępne też poprzez [index]:

```
arr[2] = arr[0] + arr[1]
```

Tablice specjalizowane (wydajność!):
ByteArray, IntArray, CharArray, DoubleArray, etc.

Kolekcje mutowalne i niemutowalne

List (interfejs) -- lista niemutowalna (zawiera size, get etc.).

Podobnie Set, Map.

```
val numbers: MutableList<Int> = mutableListOf(1, 2, 3)
val readOnlyView: List<Int> = numbers
println(numbers)       // prints "[1, 2, 3]"
numbers.add(4)
println(readOnlyView)   // prints "[1, 2, 3, 4]"
readOnlyView.clear()    // COMPILE ERROR!
```

Tworzenie kolekcji: metody takie jak listOf, mutableListOf, setOf, mutableSetOf, etc.
Inicjalizacja słownika: mapOf(a to b, c to d).

Typy niemutowalne (read-only) są w Kotlinie kowariantne

Kowariancja – forma polimorfizmu. Obiekt możemy przypisać do zmiennej tego samego typu lub dowolnego typu bazowego

Przypomnijmy:

w Javie List<Integer> NIE JEST podtypem List<Number>.

W Kotlinie JEST!

Dotyczy tylko typów niemutowanych (m.in. List).

Dla MutableList już nie ma takiego dziedziczenia.

Adnotacje kowariantności / kontrawariantności: słowa kluczowe in, out

Poniżej: typ Source jest kowariantny względem parametru T,
tzn. Source jest producentem obiektów typu T, a nie ich konsumentem.
Czytaj: Source nie posiada metod, które mają T jako parametr (natomiast mogą zwracać T).

```
interface Source<out T> {
    fun nextT(): T
}

fun demo(strs: Source<String>) {
    val objects: Source<Any> = strs  // OK, since T is an out-param
    // ...
}
```

Obiekt typu Source<String> może tylko produkować obiekty typu String,
a każdy String jest (oczywiście) obiektem typu Any.

Adnotacje kowariantności / kontrawariantności: słowa kluczowe in, out (c.d.)

Poniżej: typ Comparable jest kontrawariantny względem param. T, tzn. Comparable jest konsumentem obiektów typu T, a nie ich producentem.

```
interface Comparable<in T> {
    operator fun compareTo(other: T): Int
}

fun demo(x: Comparable<Number>) {
    x.compareTo(1.0) // 1.0 has type Double (a subtype of Number)
    // --> can assign x to a variable of type Comparable<Double>
    val y: Comparable<Double> = x  // OK!
}
```

x jest typu Comparable<Number>, więc akceptuje porównania z dowolnymi liczbami (Number). W szczególności więc akceptuje porównania z typem Double.

Klasy danych (*data classes*)

(Prawie) odpowiednik case class w Scali.

Jeśli głównym celem danej klasy jest przechowywanie danych, to możemy napisać np.:

data class User(val name: String, val age: Int)

Z automatu otrzymujemy metody: equals, hashCode, toString, componentN (→ patrz nast. slajd), copy.

Uwaga: do generacji equals etc. kompilator używa tylko właściwości z konstruktora głównego.

```
data class Person(val name: String) {
   var age: Int = 32  // equals, toString etc. tego nie użyją
}
val p = Person("Stefan"); println(p)  // Person(name=Stefan)
```

Klasy danych, c.d.

Przykład podobny do poprzedniego:

```
data class Person(name: String) {
   var age: Int = 32  // equals, toString etc. tego nie użyją
}
val p = Person("Stefan"); println(p)  //  ?
```

Błąd kompilacji:

Data class primary constructor must have only property (val / var) parameters

W Scali inaczej: pominięcie val/var w case class → val
(jako słowo domyślne).

Dekonstrukcja

```
val (name, age) = person
```

→ skrót względem następującego kodu:

```
val name = person.component1()
val age = person.component2()
```

Metody component1 itd. są
automatycznie generowane w klasach danych (*data classes*).

Możemy mieć więc następującą klasę:

```
data class Result(val result: Int, val status: Status)
 fun f(...): Result {
  // computations
  return Result(result, status)
 }

 ...
 val (result, status) = f(...)
```

Referencje przyjmujące i nieprzyjmujące null

```
var a: String = "abc"
a = null // compilation error

var b: String? = "abc"
b = null // ok
```

UWAGA:
chcemy wyciągnąć tekst z pola typu EditText (Android):

```
// Incorrect (returns the "null" string!):
val text = view.textField?.text.toString() ?: ""

// Correct:
val text = view.textField?.text?.toString() ?: ""
```

Jeszcze o null

Sufiks !! ignoruje info o możliwym nullu:

```
val message: String? = null
println(message!!)  // rzucany KotlinNullPointerException
```

Filtracja nulli:

```
val a: List<Int?> = listOf(1, 2, 3, null)
val b: List<Int> = a.filterNotNull()
```

Operator „Elvis” (?:)

```
val value: String = data?.first() ?: "Nic"

// jeśli data lub data.first() jest null, to pod value podstawi "Nic”,
// w przeciwnym razie podstawi (oczywiście) data.first()
```

Czy ta funkcja jest poprawna?

```
fun factorial(i: Int): Int {
  if (i <= 1)
    return 1
  else
    return i * factorial(i - 1)
}
```

```
// BETTER:
fun factorial(i: Int): Long {
  if (i < 0)
    throw IllegalArgumentException("Factorial of
      negative ints undefined!")
  return if (i <= 1) 1L else i * factorial(i - 1)
}
```

niekonieczne tutaj, ale dla czytelności

Zakresy (*ranges*)

```
if (i in 1..10) {  // equivalent of 1 <= i && i <= 10
   println(i)
}
for (i in 4 downTo 1) print(i)
for (i in 1..4 step 2) print(i)
for (i in 1 until 10) print(i)   // [1, 10), without 10
```

Elementy składniowe:

.. (wywołuje funkcję rangeTo)

in

!in

step

downTo

until

Działają z użyciem interfejsu ClosedRange<T>
oraz klas IntProgression, CharProgression etc.

Notacja infiksowa

Czym jest np. 4 downTo 1
z poprzedniego slajdu?

Tak, to skrót od 4.downTo(1).

W uproszczeniu: wywołania funkcji zdefiniowanych ze słowem kluczowym infix, z jednym parametrem,
który nie ma wartości domyślnej,
mogą być wywoływane w postaci infiksowej.
Czyli bez kropki bez nawiasów.

```
infix fun Int.shl(x: Int): Int { ... }
println(1 shl 3)  // 8
```

Sprytne rzutowania (*smart casts*)

```
fun printStringLength(any: Any) {
    if (any is String) {
        println(any.length)
    }
}
```

W Javie analogicznie:

```
if (any instanceof String)
  System.out.println( ((String)any).length() );
```

```
// Kotlin, inny przykład:
fun isEmptyString(any: Any): Boolean {
    return any is String && any.length == 0
}
```

Funkcje rozszerzające (*extension functions*)

Koncepcja znana z C#: do istniejącej klasy
(nawet finalnej) można dodać funkcję.

```
fun String.lastChar(): Char = this[this.length - 1]
println("abc".lastChar())   // 'c'
```

Podobny mechanizm:
rozszerzenia właściwości (*extension properties*).

```
val ViewGroup.children: List
    get() = (0..childCount -1).map { getChildAt(it) }
```

Rozszerzona właściwość NIE zachowuje stanu
(musi korzystać z istniejących metod, aby odczytać lub zmodyfikować stan obiektu).

Extension properties, inny przykład

```
val List<Int>.getEven: List<Int>
  get() = (0..this.size-1 step 2).map { get(it) }

fun main()
{
    val aList = listOf(1, 4, 9, 16, 25, 36)
    println(aList.getEven)
}
```

Rozszerzonych właściwości
nie możemy zdefiniować lokalnie
(np. w funkcji main).

Pasowanie wzorców

(*pattern matching*)

Słowo when. Skromniejsze możliwości niż w Scali.

```
val greeting = when (x) {
    "English" -> "How are you?"
    "German" -> "Wie geht es dir?"
    else -> "I don't know that language yet :("
}
print(greeting)
```

```
val names = listOf("John", "Sarah", "Tim", "Maggie")
when (x) {
    in names -> print("I know that name!")
    !in 1..10 -> print("Argument was not in the range from 1 to 10")
    is String -> print(x.length) // smart casting!
}
```

Współprogramy

(*coroutines*)

https://github.com/Kotlin/kotlinx.coroutines/blob/master/docs/basics.md#your-first-coroutine

```
import kotlinx.coroutines.*

fun main() {
    GlobalScope.launch { // launch new coroutine in background and continue
        delay(1000L) // non-blocking delay for 1 second
        println("World!") // print after delay
    }
    println("Hello,") // main thread continues while coroutine is delayed
    Thread.sleep(2000L) // block main thread for 2 seconds to keep JVM alive
}
```

Efekt (w tej kolejności):
Hello,
World!