

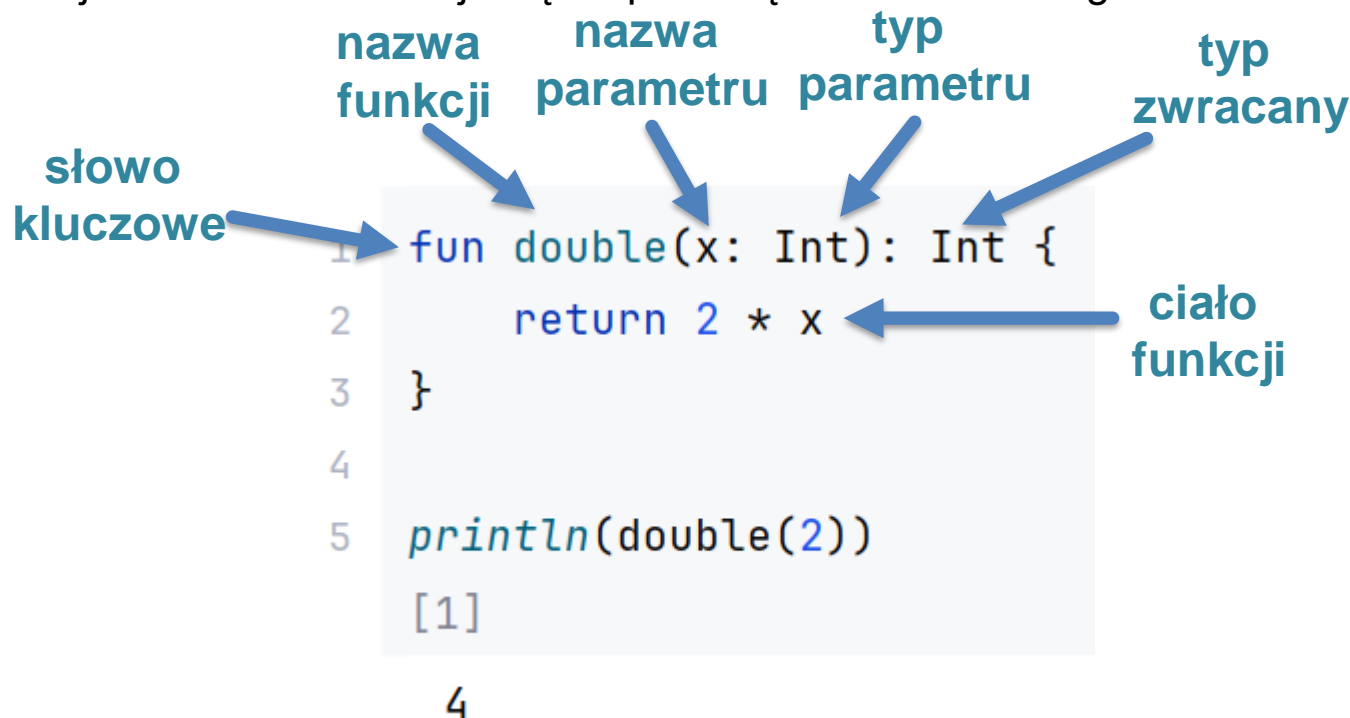


# PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 1

## WYKŁAD 2

- Funkcje

Funkcje w Kotlinie deklaruje się za pomocą słowa kluczowego **fun**.



```
1 fun double(x: Int): Int {  
2     return 2 * x  
3 }  
4  
5 println(double(2))  
[1]
```

The diagram illustrates the components of the Kotlin function declaration `fun double(x: Int): Int { ... }` with the following labels and arrows:

- słowo kluczowe** (keyword) points to `fun`.
- nazwa funkcji** (function name) points to `double`.
- nazwa parametru** (parameter name) points to `x`.
- typ parametru** (parameter type) points to `Int`.
- typ zwracany** (return type) points to `Int`.
- ciało funkcji** (function body) points to the code block `{ return 2 * x }`.

Funkcja **double** przyjmuje jeden parametr **x** typu **Int** i zwraca wartość typu **Int**. Wartość zwrotna jest deklarowana po dwukropku `:` w nagłówku funkcji.

Parametry funkcji są definiowane w formacie: **nazwa: typ**. Mogą one być oddzielane przecinkami, a każdy z parametrów musi mieć jawnie określony typ.

słowo  
kluczowe

	nazwa funkcji	nazwa parametru	typ parametru	wartość domyślna
1	fun	greet		
2		name	String	"Rafał"
3		greeting	String	"Hello"
4	)			
5	{			
6				
7				
8				
9				
10				

```
1 fun greet(  
2     name: String = "Rafał",  
3     greeting: String = "Hello"  
4 ) {  
5     println("$greeting, $name!")  
6 }  
7  
8 greet("Anna")  
9 greet("Bob", "Welcome")  
10 greet()  
[9]
```

Hello, Anna!

Welcome, Bob!

Hello, Rafał!

```
1 fun orderCoffee(  
2     size: String = "Medium",  
3     type: String = "Latte",  
4     sugar: Int = 1,  
5     milk: Boolean = true,  
6     extraShot: Boolean = false,  
7 ) {
```

```
12 }
```

```
13
```

```
14 orderCoffee(  
15     size = "Small",  
16     type = "Espresso",  
17     sugar = 0,  
18     milk = false,  
19     extraShot = true  
20 )
```

} argumenty  
nazwane

argumenty  
zmienne



```
1 ✓ fun printAll(vararg items: String) {  
2 ✓     for (item in items) {  
3         println(item)  
4     }  
5 }  
6  
7 printAll("Apple", "Banana", "Cherry")  
[23]
```

Apple  
Banana  
Cherry

# Funkcje rozszerzające

**Funkcje rozszerzające** pozwalają na dodawanie nowych funkcji do **istniejących klas**, bez konieczności ich modyfikacji. Dzięki nim możemy rozszerzyć funkcjonalność klas standardowych (np. **String**, **List**) lub klas własnych.

Funkcje rozszerzające są definiowane **poza klasą**, ale mają dostęp do **obiektu klasy**, dla której zostały utworzone, za pomocą słowa kluczowego **this**.

```
fun Typ.obiektowy.nazwaFunkcji(parametry): TypZwracany {  
    // ciało funkcji  
}
```

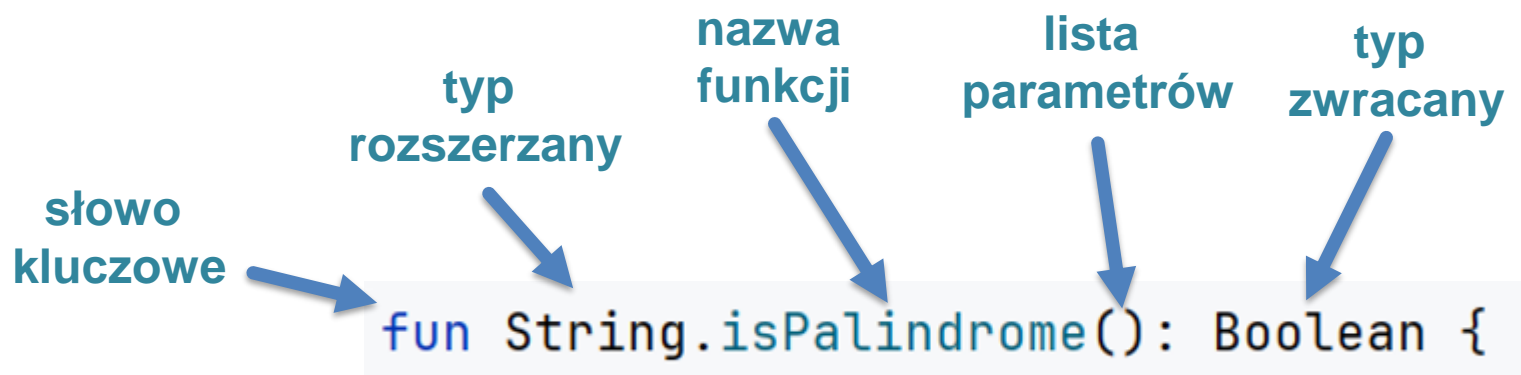


Diagram illustrating the components of the function signature `fun String.isPalindrome(): Boolean`:

- słowo kluczowe** points to `fun`
- typ rozszerzany** points to `String`
- nazwa funkcji** points to `isPalindrome`
- lista parametrów** points to `()`
- typ zwracany** points to `Boolean`

**Funkcje rozszerzające** pozwalają na dodawanie nowych funkcji do **istniejących klas**, bez konieczności ich modyfikacji. Dzięki nim możemy rozszerzyć funkcjonalność klas standardowych (np. **String**, **List**) lub klas własnych.

Funkcje rozszerzające są definiowane **poza klasą**, ale mają dostęp do **obiektu klasy**, dla której zostały utworzone, za pomocą słowa kluczowego **this**.

```
fun Typ.obiektowy.nazwaFunkcji(parametry): TypZwracany {  
    // ciało funkcji  
}
```

```
1 fun String.isPalindrome(): Boolean {  
2     val cleaned = this.replace("\\s".toRegex(), "").lowercase()  
3     return cleaned == cleaned.reversed()  
4 }  
5  
6 val word = "Kajak"  
7 println(word.isPalindrome())  
8 println("ABBA".isPalindrome())  
[27]  
  
true  
true
```

# Funkcje infiksowe

Funkcje oznaczone słowem kluczowym **infix** mogą być wywoływane przy użyciu **notacji infiksowej**, co oznacza, że **można pominąć kropkę i nawiasy w wywołaniu**.

Aby funkcja mogła być oznaczona jako infiksowa, musi spełniać poniższe wymagania:

- **Funkcja musi być funkcją członkowską lub funkcją rozszerzającą.**
- **Nie można używać funkcji globalnych jako infiksowych.**
- **Funkcja musi mieć dokładnie jeden parametr.**

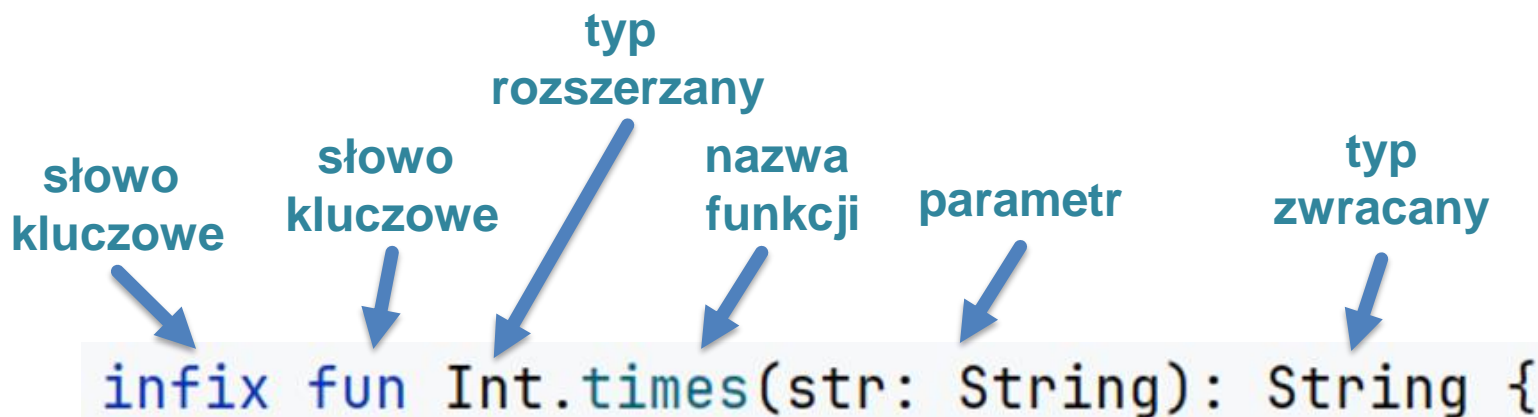


Diagram illustrating the components of the infix function declaration `infix fun Int.times(str: String): String {`:

- `infix`: słowo kluczowe
- `fun`: słowo kluczowe
- `Int`: typ rozszerzany
- `times`: nazwa funkcji
- `(str: String)`: parametr
- `: String`: typ zwracany



Funkcje oznaczone słowem kluczowym **infix** mogą być wywoływane przy użyciu **notacji infiksowej**, co oznacza, że **można pominąć kropkę i nawiasy w wywołaniu**.

Aby funkcja mogła być oznaczona jako infiksowa, musi spełniać poniższe wymagania:

- **Funkcja musi być funkcją członkowską lub funkcją rozszerzającą.**
- **Nie można używać funkcji globalnych jako infiksowych.**
- **Funkcja musi mieć dokładnie jeden parametr.**

```
1 ✓ infix fun Int.times(str: String): String {  
2     return str.repeat(this)  
3 }  
4  
5 val result = 3 times "Hello "  
6 println(result)  
[32]
```

Hello Hello Hello

Funkcje lambda to zwięzły sposób definiowania **funkcji anonimowych**. Są często wykorzystywane jako funkcje przekazane jako argumenty do innej funkcji.

```
{ [parametry] -> [ciało] }
```

Funkcje lambda to zwięzły sposób definiowania **funkcji anonimowych**. Są często wykorzystywane jako funkcje przekazane jako argumenty do innej funkcji.

```
{ [parametry] -> [ciało] }
```

lista  
parametrów

ciało  
funkcji

```
1 val sum = { a: Int, b: Int -> a + b }  
2 println(sum(3, 5)) // Wynik: 8  
[64]
```

8

Funkcje lambda to zwięzły sposób definiowania **funkcji anonimowych**. Są często wykorzystywane jako funkcje przekazane jako argumenty do innej funkcji.

```
{ [parametry] -> [ciało] }
```

```
1 val sum = { a: Int, b: Int -> a + b }  
2 println(sum(3, 5)) // Wynik: 8  
[64]
```

8

Dla lambd z jednym parametrem można pominąć nazwę parametru i użyć domyślnego słowa kluczowego **it**

```
1 val numbers = listOf(1, 2, 3, 4, 5)  
2  
3 // Lambda z 'map'  
4 val squared = numbers.map { it * it }  
5 println(squared) // Wynik: [1, 4, 9, 16, 25]  
[7]
```

[1, 4, 9, 16, 25]

Lambdy zwracają wartość **ostatniego wyrażenia**.

lista parametrów

wartość zwracana

ciało funkcji

```
1 val subtract = { a: Int, b: Int ->
2     println("Odejmowanie $b od $a")
3     a - b ^lambda
4 }
5 println(subtract(10, 3)) // Wynik: Odejmowanie 3 od 10
[12]
```

Odejmowanie 3 od 10  
7

Lambdy to **wyrażenia funkcjonalne**, które można przekazywać jako parametry do funkcji.

```
1 fun repeatAction(times: Int, action: (String, Int) -> Unit) {  
2     for (i in 1 ≤ .. ≤ times) {  
3         action("Parametr: ", i)  
4     }  
5 }
```

Lambdy to **wyrażenia funkcyjne**, które można przekazywać jako parametry do funkcji.

słowo  
kluczowe

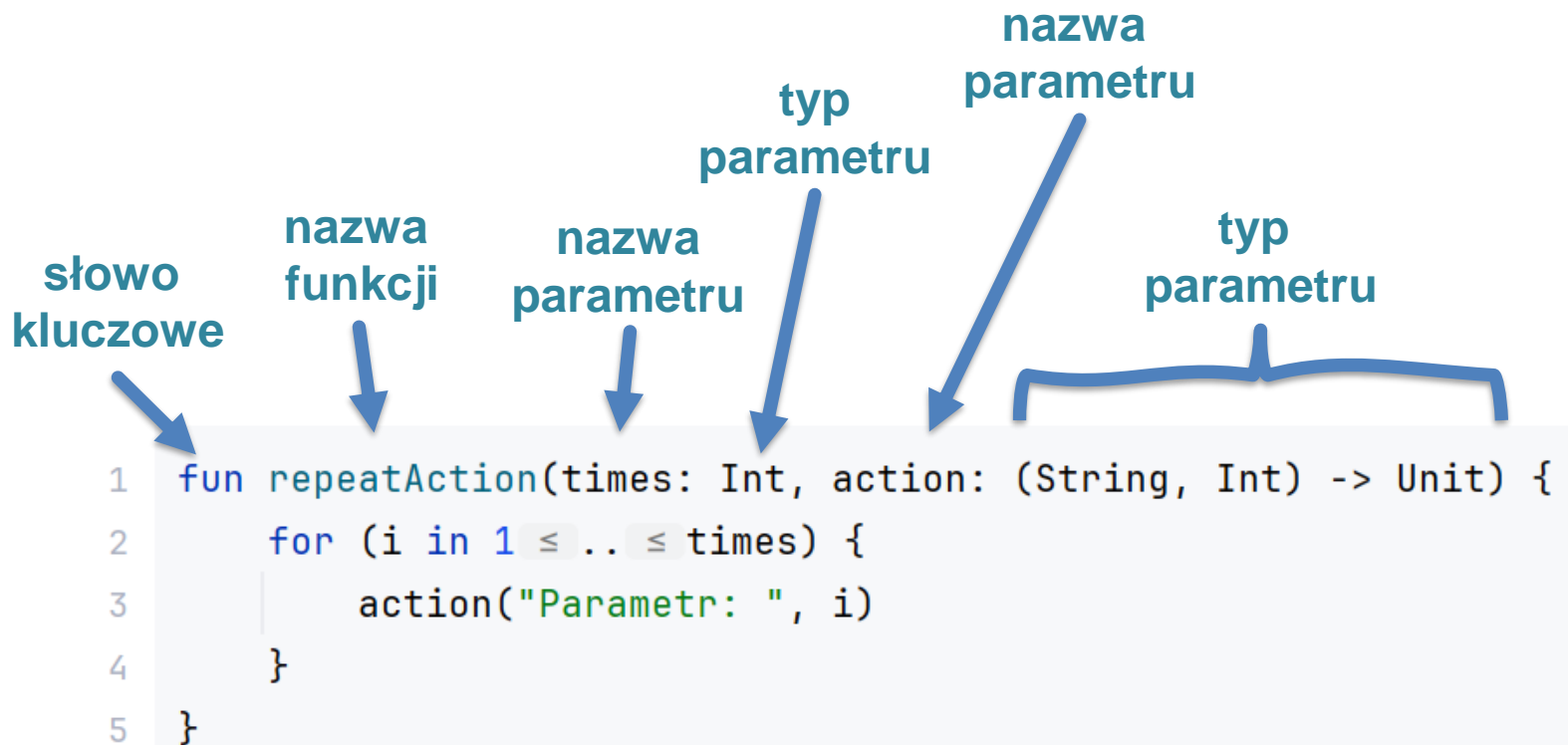
nazwa  
funkcji

nazwa  
parametru

typ  
parametru

```
1 fun repeatAction(times: Int, action: (String, Int) -> Unit) {  
2     for (i in 1 ≤ .. ≤ times) {  
3         action("Parametr: ", i)  
4     }  
5 }
```

Lambdy to **wyrażenia funkcyjne**, które można przekazywać jako parametry do funkcji.



```
1 fun repeatAction(times: Int, action: (String, Int) -> Unit) {
2   for (i in 1 ≤ .. ≤ times) {
3     action("Parametr: ", i)
4   }
5 }
```

słowo  
kluczowe

nazwa  
funkcji

nazwa  
parametru

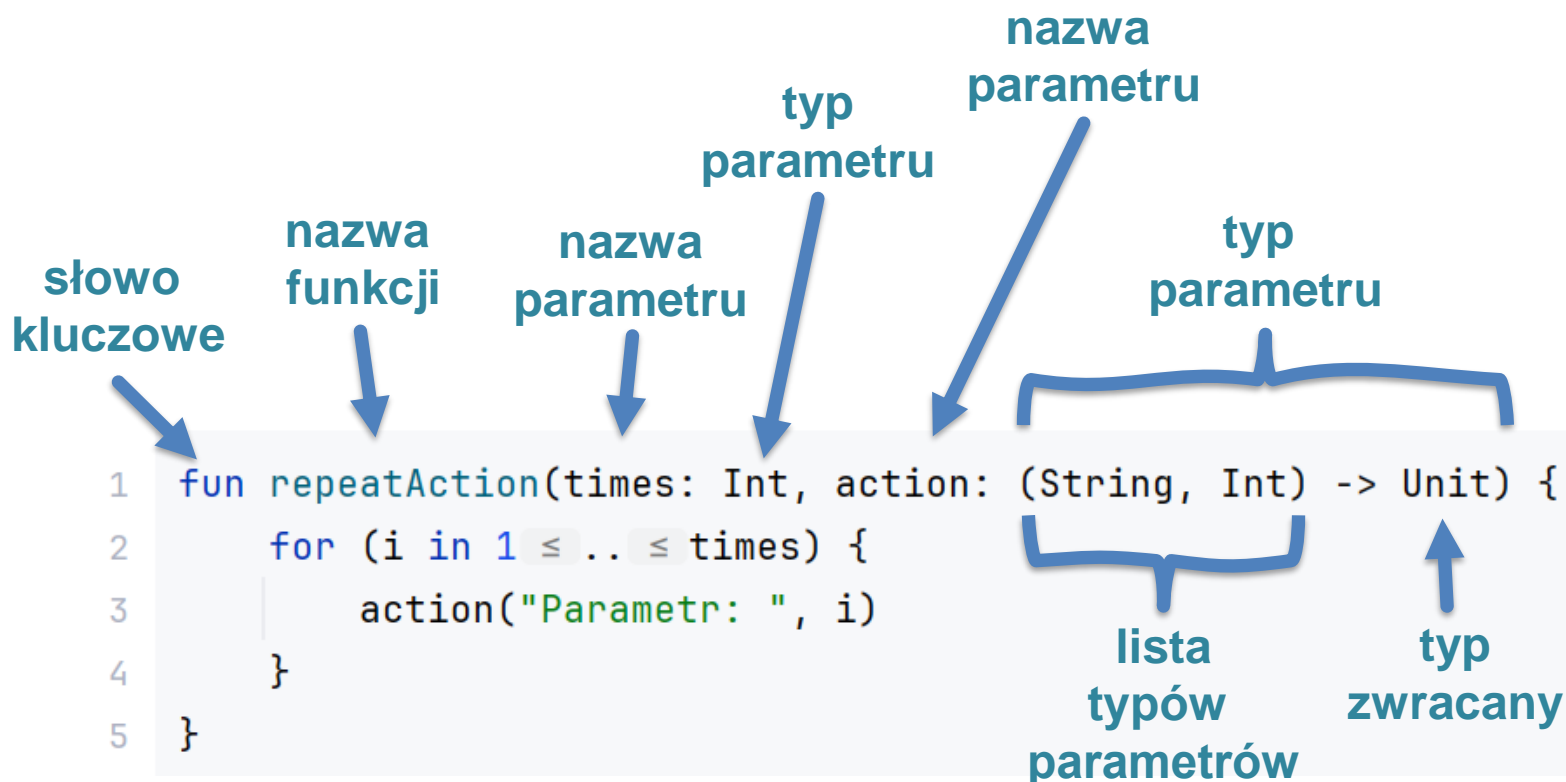
typ  
parametru

nazwa  
parametru

typ  
parametru



Lambdy to **wyrażenia funkcjonalne**, które można przekazywać jako parametry do funkcji.



```
1 fun repeatAction(times: Int, action: (String, Int) -> Unit) {  
2   for (i in 1 ≤ .. ≤ times) {  
3     action("Parametr: ", i)  
4   }  
5 }
```

słowo  
kluczowe

nazwa  
funkcji

nazwa  
parametru

typ  
parametru

nazwa  
parametru

typ  
parametru

lista  
typów  
parametrów

typ  
zwracany

Lambdy to **wyrażenia funkcjonalne**, które można przekazywać jako parametry do funkcji.

```
fun repeatAction(times: Int, action: (String, Int) -> Unit) {
```

```
repeatAction(3, {s, i -> println("$s$i")})
```

Parametr: 1

Parametr: 2

Parametr: 3

Lambdy to **wyrażenia funkcjonalne**, które można przekazywać jako parametry do funkcji.

```
fun repeatAction(times: Int, action: (String, Int) -> Unit) {
```

```
repeatAction(3, {s, i -> println("$s$i")})
```

Parametr: 1

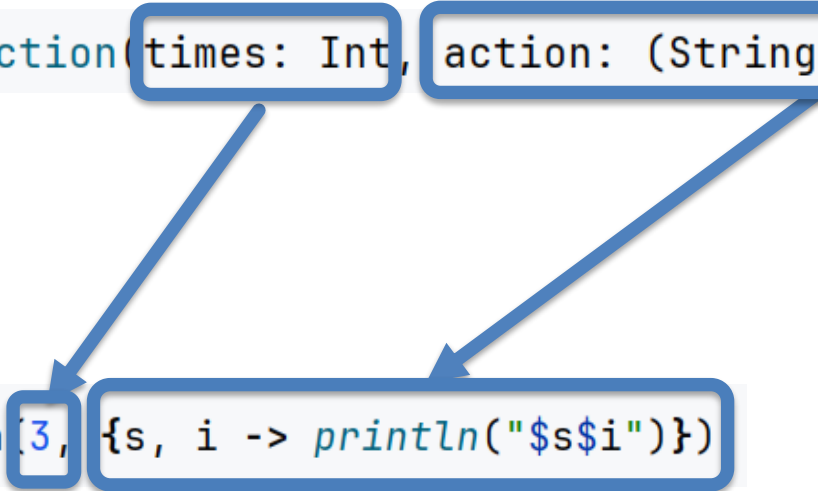
Parametr: 2

Parametr: 3

Lambdy to **wyrażenia funkcjonalne**, które można przekazywać jako parametry do funkcji.

```
fun repeatAction(times: Int, action: (String, Int) -> Unit) {
```

```
repeatAction(3, {s, i -> println("$s$i")})
```



Parametr: 1

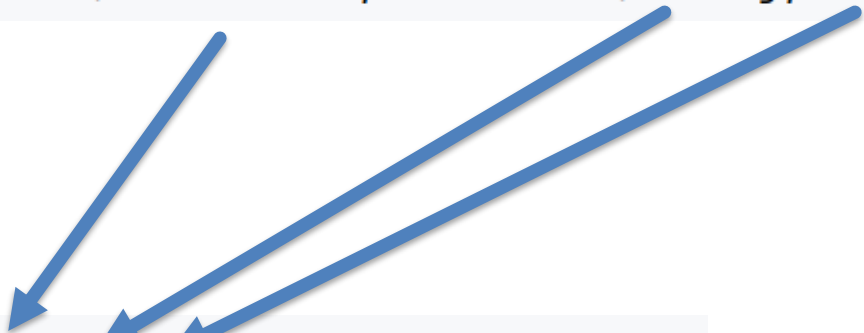
Parametr: 2

Parametr: 3

Lambdy to **wyrażenia funkcjonalne**, które można przekazywać jako parametry do funkcji.

```
fun repeatAction(times: Int, action: (String, Int) -> Unit) {
```

```
repeatAction(3, {s, i -> println("$s$i")})
```



Parametr: 1

Parametr: 2

Parametr: 3

Lambdy to **wyrażenia funkcjonalne**, które można przekazywać jako parametry do funkcji.

```
1 ✓ fun repeatAction(times: Int, action: (String, Int) -> Unit) {  
2   for (i in 1 ≤ .. ≤ times) {  
3     action("Parametr: ", i)  
4   }  
5 }
```

```
8 repeatAction(3, {s, i -> println("$s$i")})  
9
```

✓ [4] 201ms

Parametr: 1

Parametr: 2

Parametr: 3

Jeśli lambda jest ostatnim parametrem funkcji, można ją zapisać **poza nawiasami**

```
1 ✓ fun performOperation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {  
2     return operation(a, b)  
3 }  
4  
5 val result1 = performOperation(10, 5, { x, y -> x + y })  
6  
7  
8  
9 val result2 = performOperation(10, 5) { x, y -> x + y }  
10 println(result1)  
11 println(result2)  
✓ [2] 159ms
```

15

15

Jeśli lambda jest ostatnim parametrem funkcji, można ją zapisać **poza nawiasami**

```
1 fun performOperation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {  
2     return operation(a, b)  
3 }  
4  
5 val result1 = performOperation(10, 5, { x, y -> x + y })  
6  
7  
8  
9 val result2 = performOperation(10, 5) { x, y -> x + y }  
10 println(result1)  
11 println(result2)  
✓ [2] 159ms
```

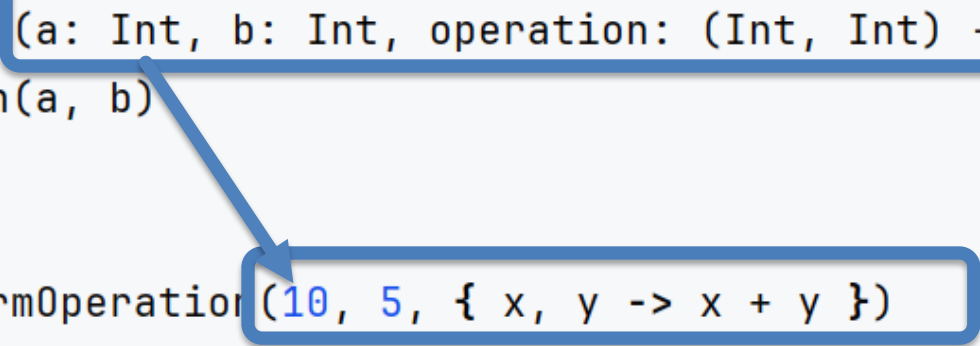
15

15



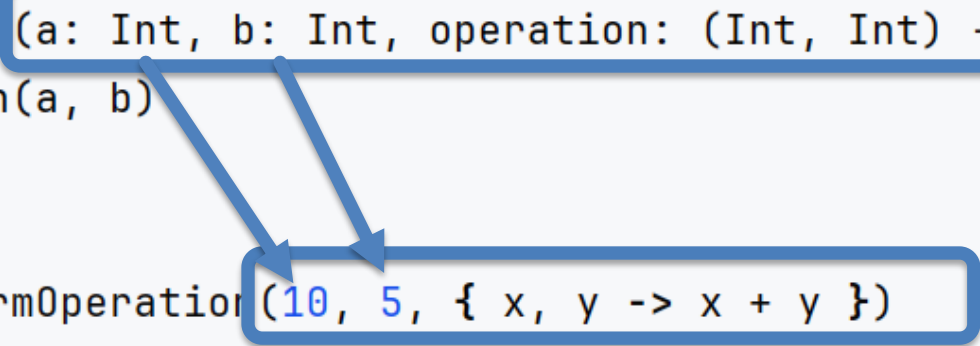
Jeśli lambda jest ostatnim parametrem funkcji, można ją zapisać **poza nawiasami**

```
1 fun performOperation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {  
2     return operation(a, b)  
3 }  
4  
5 val result1 = performOperation(10, 5, { x, y -> x + y })  
6  
7  
8  
9 val result2 = performOperation(10, 5) { x, y -> x + y }  
10 println(result1)  
11 println(result2)  
✓ [2] 159ms
```



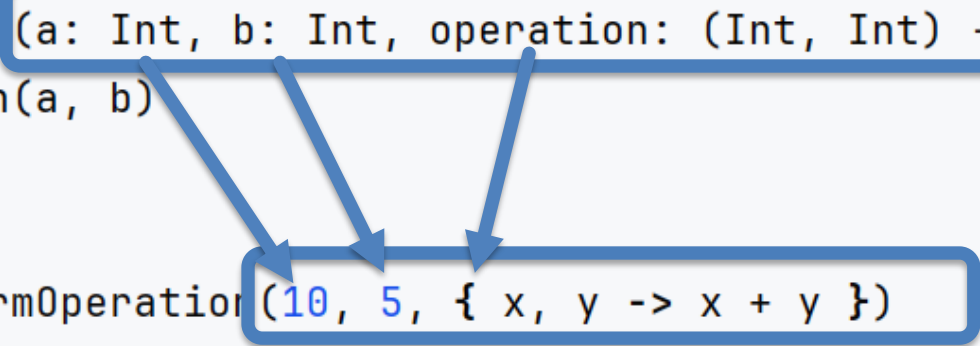
Jeśli lambda jest ostatnim parametrem funkcji, można ją zapisać **poza nawiasami**

```
1 fun performOperation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {  
2     return operation(a, b)  
3 }  
4  
5 val result1 = performOperation(10, 5, { x, y -> x + y })  
6  
7  
8  
9 val result2 = performOperation(10, 5) { x, y -> x + y }  
10 println(result1)  
11 println(result2)  
✓ [2] 159ms
```



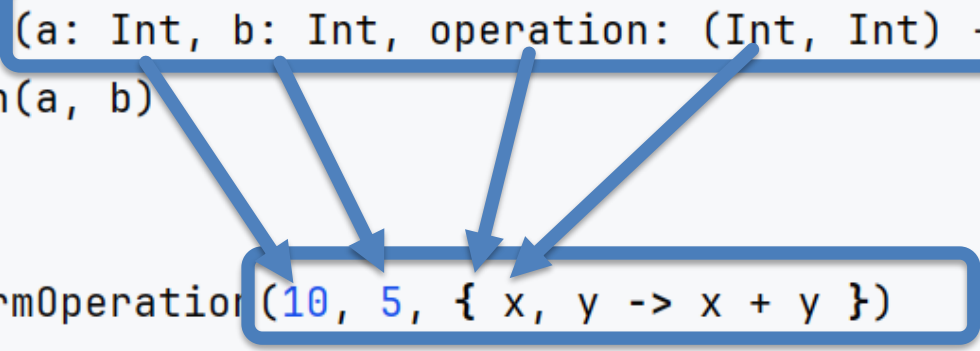
Jeśli lambda jest ostatnim parametrem funkcji, można ją zapisać **poza nawiasami**

```
1 fun performOperation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {  
2     return operation(a, b)  
3 }  
4  
5 val result1 = performOperation(10, 5, { x, y -> x + y })  
6  
7  
8  
9 val result2 = performOperation(10, 5) { x, y -> x + y }  
10 println(result1)  
11 println(result2)  
✓ [2] 159ms
```



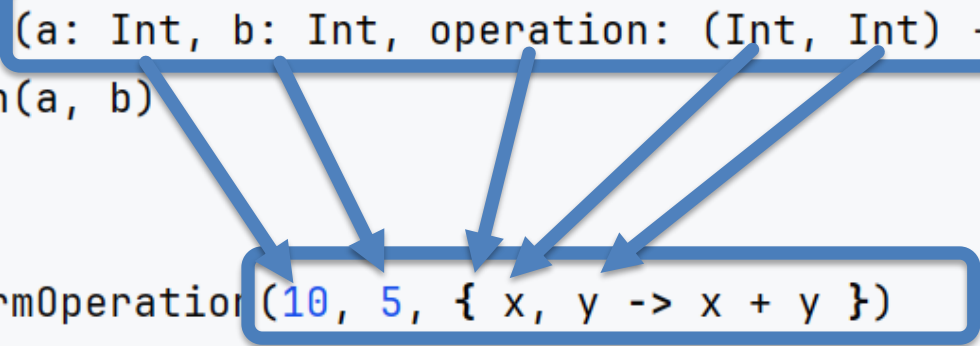
Jeśli lambda jest ostatnim parametrem funkcji, można ją zapisać **poza nawiasami**

```
1 fun performOperation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {  
2     return operation(a, b)  
3 }  
4  
5 val result1 = performOperation(10, 5, { x, y -> x + y })  
6  
7  
8  
9 val result2 = performOperation(10, 5) { x, y -> x + y }  
10 println(result1)  
11 println(result2)  
✓ [2] 159ms
```



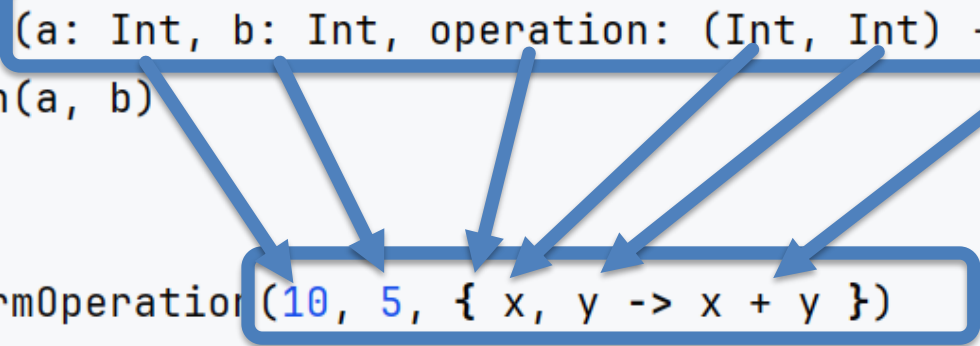
Jeśli lambda jest ostatnim parametrem funkcji, można ją zapisać **poza nawiasami**

```
1 fun performOperation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {  
2     return operation(a, b)  
3 }  
4  
5 val result1 = performOperation(10, 5, { x, y -> x + y })  
6  
7  
8  
9 val result2 = performOperation(10, 5) { x, y -> x + y }  
10 println(result1)  
11 println(result2)  
✓ [2] 159ms
```



Jeśli lambda jest ostatnim parametrem funkcji, można ją zapisać **poza nawiasami**

```
1 fun performOperation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {  
2     return operation(a, b)  
3 }  
4  
5 val result1 = performOperation(10, 5, { x, y -> x + y })  
6  
7  
8  
9 val result2 = performOperation(10, 5) { x, y -> x + y }  
10 println(result1)  
11 println(result2)  
✓ [2] 159ms
```



The diagram illustrates the mapping of arguments from the function call to the function definition. Blue arrows point from the arguments in the function call to the corresponding parameters in the function definition. Specifically, arrows point from '10' to 'a', '5' to 'b', and the lambda expression '{ x, y -> x + y }' to 'operation'. The function signature and the lambda expression are highlighted with blue boxes.

Jeśli lambda jest ostatnim parametrem funkcji, można ją zapisać **poza nawiasami**

```
1 fun performOperation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {  
2     return operation(a, b)  
3 }  
4  
5 val result1 = performOperation(10, 5, { x, y -> x + y })  
6  
7  
8  
9 val result2 = performOperation(10, 5) { x, y -> x + y }  
10 println(result1)  
11 println(result2)  
✓ [2] 159ms
```

15

15

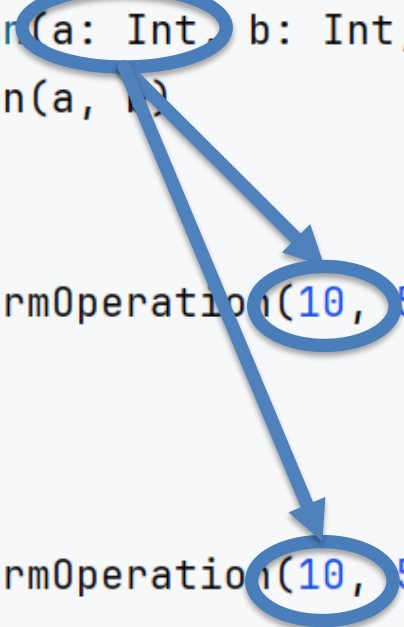
Jeśli lambda jest **ostatnim parametrem funkcji**, można ją zapisać **poza nawiasami**

```
1 fun performOperation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {  
2     return operation(a, b)  
3 }  
4  
5 val result1 = performOperation(10, 5, { x, y -> x + y })  
6  
7  
8  
9 val result2 = performOperation(10, 5) { x, y -> x + y }  
10 println(result1)  
11 println(result2)  
✓ [2] 159ms
```



Jeśli lambda jest ostatnim parametrem funkcji, można ją zapisać **poza nawiasami**

```
1 fun performOperation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {  
2     return operation(a, b)  
3 }  
4  
5 val result1 = performOperation(10, 5, { x, y -> x + y })  
6  
7  
8  
9 val result2 = performOperation(10, 5) { x, y -> x + y }  
10 println(result1)  
11 println(result2)  
✓ [2] 159ms
```

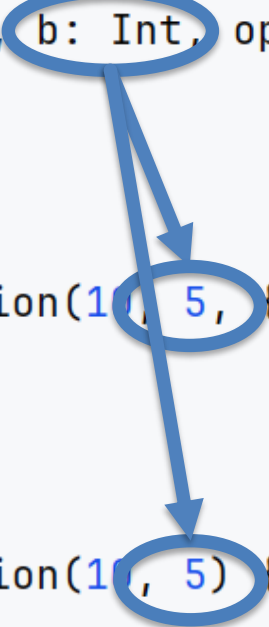


15

15

Jeśli lambda jest ostatnim parametrem funkcji, można ją zapisać **poza nawiasami**

```
1 fun performOperation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {  
2     return operation(a, b)  
3 }  
4  
5 val result1 = performOperation(1, 5, { x, y -> x + y })  
6  
7  
8  
9 val result2 = performOperation(1, 5) { x, y -> x + y }  
10 println(result1)  
11 println(result2)  
✓ [2] 159ms
```

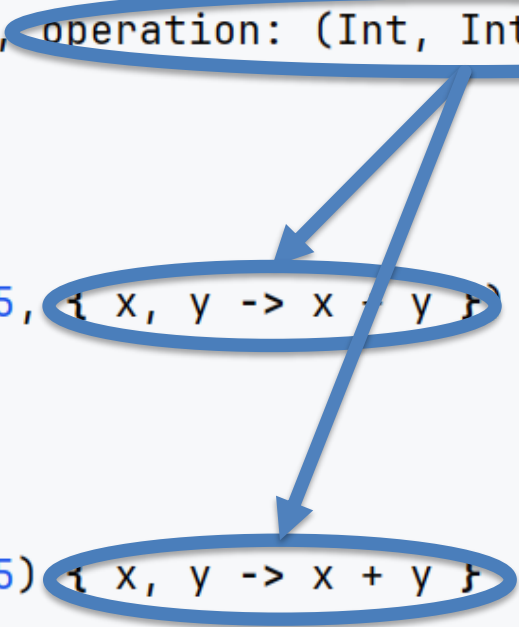


15

15

Jeśli lambda jest ostatnim parametrem funkcji, można ją zapisać **poza nawiasami**

```
1 fun performOperation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {  
2     return operation(a, b)  
3 }  
4  
5 val result1 = performOperation(10, 5, { x, y -> x - y })  
6  
7  
8  
9 val result2 = performOperation(10, 5, { x, y -> x + y })  
10 println(result1)  
11 println(result2)  
✓ [2] 159ms
```



Jeśli lambda jest ostatnim parametrem funkcji, można ją zapisać **poza nawiasami**


```
1 ✓ fun performOperation(a: Int, b: Int, operation: (Int, Int) -> Int): Int {  
2     return operation(a, b)  
3 }  
4  
5 val result1 = performOperation(10, 5, { x, y -> x + y })  
6  
7  
8  
9 val result2 = performOperation(10, 5) { x, y -> x + y }  
10 println(result1)  
11 println(result2)  
✓ [2] 159ms
```

15

15

Jeśli lambda jest ostatnim parametrem funkcji, można ją zapisać **poza nawiasami**

```
1  val numbers = listOf(1, 2, 3, 4, 5)
2  // Lambda z 'filter'
3  val evens = numbers.filter { it % 2 == 0 }
4  println(evens) // Wynik: [2, 4]
[42]
```



[2, 4]

Przy pracy z funkcjami wyższego rzędu, często wykorzystuje się **parametry nazwane**.

```
1 fun processFunctions(  
2     x: Int,  
3     y: Int,  
4     operation1: (Int, Int) -> Int,  
5     operation2: (Int) -> Int  
6 ): Int {  
7     val intermediateResult = operation1(x, y)  
8     return operation2(intermediateResult)  
9 }  
10  
11 val result = processFunctions(  
12     x = 10,  
13     y = 5,  
14     operation1 = { a, b -> a + b },  
15     operation2 = { it * 2 }  
16 )
```

Jeśli lambda bezpośrednio mapuje dane wejściowe na wyjście, można użyć **operatora ::**

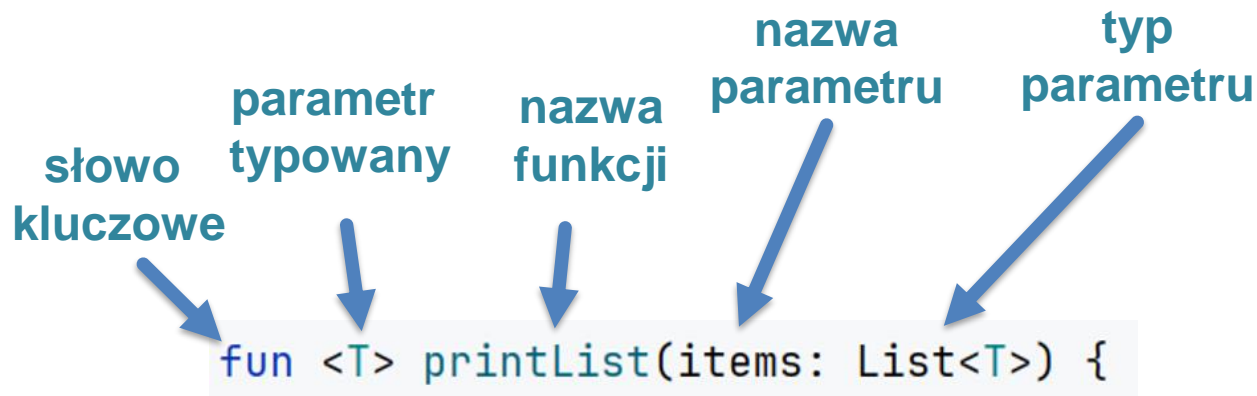
```
1 val names = listOf("Anna", "Bob", "Cleo")
2 val nameLengths = names.map(String::length)
3 println(nameLengths) // Wynik: [4, 3, 4]
```

```
[4, 3, 4]
```

**Operator ::** w Kotlinie jest referencją do funkcji lub właściwości. Używany jest do przekazywania funkcji lub właściwości jako argumentu lub do uzyskiwania referencji do nich.

**Funkcje generyczne** to takie, które przyjmują **parametry typowane**. Pozwalają one definiować funkcję, która będzie działać dla różnych typów danych bez konieczności tworzenia osobnych wersji dla każdego typu.

```
fun <T> nazwaFunkcji(argument: T): T {  
    // ciało funkcji  
}
```



- **<T>** – oznacza **parametr typu** (nazwa jest dowolna, ale często używa się **T**, **E**, **K**, **V**, **A**).
- **T** może być dowolnym typem, takim jak **Int**, **String**, czy niestandardowe klasy.



```
1 ✓ fun <T> printList(items: List<T>) {  
2 ✓     for (item in items) {  
3         print("$item ")  
4     }  
5 }  
6  
7 val intList = listOf(1, 2, 3)  
8 val stringList = listOf("a", "b", "c")  
9  
10 printList(intList)  
11 println()  
12 printList(stringList)  
13
```

[51]

1 2 3

a b c

```
1 ✓ fun <T> printItem(item: T) {  
2     println("Item: $item")  
3 }  
4
```

parametr  
typowany



```
5 printItem<Int>(1)  
6 printItem<String>("a")
```

✓ [7] 97ms

Item: 1

Item: a

Czasami wymagamy, aby **parametr generyczny** był **ograniczony** do określonego typu lub jego podtypów. Używamy do tego słowa kluczowego **where** lub :

```
1 ✓ fun <T : Comparable<T>> findMax(a: T, b: T): T {  
2     return if (a > b) a else b  
3 }  
4  
5 println(findMax(10, 20))  
6 println(findMax("apple", "banana"))  
[53]
```

20

banana

Parametr generyczny **T** musi implementować interfejs **Comparable<T>**, aby można było używać operatorów porównania.

Kotlin wspiera styl **programowania funkcyjnego**, znany jako **rekursja ogonowa**. Jest to technika pozwalająca na **implementację algorytmów rekurencyjnych**, eliminując ryzyko **przepelnienia stosu**. Dzięki modyfikatorowi **tailrec** Kotlin optymalizuje wywołania rekurencyjne, przekształcając je w pętle podczas kompilacji.

Wywołanie rekurencyjne musi być **ostatnią operacją w funkcji**. Oznacza to, że po rekurencji **nie może być żadnego dodatkowego kodu**, ani bloków **try/catch/finally**.

```
1 tailrec fun factorial(n: Int, accumulator: Int = 1): Int {  
2     return if (n <= 1) accumulator else factorial(n - 1, n * accumulator)  
3 }  
4 |  
5 println(factorial(5))  
[56]
```

```
1 ✓ fun factorial(n: Int, accumulator: Int = 1): Int {  
2     var currentN = n  
3     var currentAccumulator = accumulator  
4  
5 ✓     while (currentN > 1) {  
6         currentAccumulator *= currentN  
7         currentN -= 1  
8     }  
9  
10    return currentAccumulator  
11 }  
12  
13 println(factorial(5))  
[56]
```

Funkcje **inline** to mechanizm, który pozwala na optymalizację kodu poprzez zastępowanie wywołań funkcji ich ciałem **bezpośrednio w miejscu wywołania**.

```
1 inline fun <T, R> List<T>.filterAndMap(  
2     predicate: (T) -> Boolean,  
3     transform: (T) -> R  
4 ): List<R> {  
5     val result = mutableListOf<R>()  
6     for (element in this) {  
7         if (predicate(element)) {  
8             result.add(transform(element))  
9         }  
10    }  
11    return result  
12 }  
13  
14 val numbers = listOf(1, 2, 3, 4, 5, 6)  
15  
16 val evenSquares = numbers.filterAndMap(  
17     predicate = { it % 2 == 0 },  
18     transform = { it * it }  
19 )
```

## non-local return

Funkcje **inline** zezwalają na użycie wyrażenia **return** wewnątrz funkcji lambda.

```
1 fun executeSomeOperation(word: String, operation: () -> Unit){
2     operation()
3     println("performing operation on $word")
4 }
5
6 fun processWords(vararg words: String){
7     for (word in words) {
8         executeSomeOperation(word){
9             println("saving word")
10        }
11    }
12 }
13
14 processWords("Alice", "Bob", "Cherry")
✓ [5] 164ms
```

```
1 fun executeSomeOperation(word: String, operation: () -> Unit){
2     operation()
3     println("performing operation on $word")
4 }
5
6 fun processWords(vararg words: String){
7     for (word in words) {
8         executeSomeOperation(word){
9             if (word.startsWith("c")) return
10             println("saving word")
11         }
12     }
13 }
14
15 processWords("Alice", "Bob", "Cherry", "Dylan")
✗ [8] 63ms
```

at Cell In[8], [line 9](#), column 39: 'return' is not allowed here



```
1 ✓ fun executeSomeOperation(word: String, operation: () -> Unit){
2     operation()
3     println("performing operation on $word")
4 }
5
6 ✓ fun processNewWords(vararg words: String){
7     for (word in words) {
8         executeSomeOperation(word){
9             if (word.startsWith("C")) return@executeSomeOperation
10            println("saving word")
11        }
12    }
13 }
14
15 processNewWords("Alice", "Bob", "Cherry", "Dylan")
✓ [15] 136ms
```

```
1 ✓ fun executeSomeOperation(word: String, operation: () -> Unit){  
2     operation()  
3     println("performing operation on $word")  
4 }  
5  
6 ✓ fun processNewWords(vararg words: String){  
7 ✓     for (word in words) {  
8 ✓         executeSomeOperation(word){  
9             if (word.startsWith("C")) return@executeSomeOperation  
10                println("saving word")  
11         }  
12     }  
13 }  
14  
15 processNewWords("Alice", "Bob",  
✓ [15] 136ms
```

saving word  
performing operation on Alice  
saving word  
performing operation on Bob  
performing operation on Cherry  
saving word  
performing operation on Dylan

```
1 ✓ inline fun executeSomeOperation(word: String, operation: () -> Unit){
2     operation()
3     println("performing operation on $word")
4 }
5
6 ✓ fun processNewWords(vararg words: String){
7     for (word in words) {
8         executeSomeOperation(word){
9             if (word.startsWith("C")) return
10            println("saving word")
11        }
12    }
13 }
14
15 processNewWords("Alice", "Bob", "Cherry", "Dylan")
✓ [16] 130ms
```

```
1 ✓ inline fun executeSomeOperation(word: String, operation: () -> Unit){  
2     operation()  
3     println("performing operation on $word")  
4 }  
5  
6 ✓ fun processNewWords(vararg words: String){  
7     for (word in words) {  
8         if (word.startsWith("C")) return  
9         println("saving word")  
10        println("performing operation on $word")  
11    }  
12 }  
13  
14  
15 processNewWords("Alice", "Bob", "Cherry", "Dylan")  
✓ [16] 130ms
```

```
1 ✓ inline fun executeSomeOperation(word: String, operation: () -> Unit){  
2     operation()  
3     println("performing operation on $word")  
4 }  
5  
6 ✓ fun processNewWords(vararg words: String){  
7 ✓     for (word in words) {  
8 ✓         executeSomeOperation(word){  
9             if (word.startsWith("C")) return  
10            println("saving word")  
11        }  
12    }  
13 }  
14  
15 processNewWords("Alice", "Bob",  
✓ [16] 130ms
```

saving word  
performing operation on Alice  
saving word  
performing operation on Bob

Funkcje lambda **nie mogą być zagnieżdżane**.

```
1 ✓ inline fun executeSomeOperation(word: String, operation: () -> Unit){  
2     val x = {operation()} // zagnieżdżenie  
3     x()  
4     println("performing operation on $word")  
5 }  
6  
7 ✓ fun processNewWords(vararg words: String){  
8     for (word in words) {  
9         executeSomeOperation(word){  
10             if (word.startsWith("C")) return  
11             println("saving word")  
12         }  
13     }  
14 }  
15  
16 processNewWords("Alice", "Bob", "Cherry", "Dylan")  
✗ [18] 59ms
```

at Cell In[18], [line 2](#), column 14: Can't inline 'operation' here: it

Funkcje lambda **nie mogą być zagnieżdżane**.

```
1 ✓ inline fun executeSomeOperation(word: String, operation: () -> Unit){  
2     val x = {operation()} // zagnieżdżenie  
3     x()  
4     println("performing operation on $word")  
5 }  
6  
7 ✓ fun processNewWords(vararg words: String){  
8 ✓     for (word in words) {  
9 ✓         executeSomeOperation(word){  
10             if (word.startsWith("C")) return  
11             println("saving word")  
12         }  
13     }  
14 }  
15  
16 processNewWords("Alice", "Bob", "Cherry", "Dylan")
```

Can't inline 'operation' here: it may contain non-local returns. Add 'crossinline' modifier to parameter declaration 'operation'

Występują sytuacje, w których chcemy mieć możliwość **przejścia** (ang. ***crossed***) do **innego kontekstu**. Do takich sytuacji Kotlin oferuje modyfikator **crossinline**

```
1 inline fun executeSomeOperation(word: String, crossinline operation: () ->
  Unit){
2     val x = {operation()} // zagnieżdżenie
3     x()
4     println("performing operation on $word")
5 }
6
7 fun processNewWords(vararg words: String){
8     for (word in words) {
9         executeSomeOperation(word){
10             if (word.startsWith("C")) return
11             println("saving word")
12         }
13     }
14 }
15
16 processNewWords("Alice", "Bob", "Cherry", "Dylan")
X [20] 84ms
```

at Cell In[20], line 10, column 39: 'return' is not allowed here



Występują sytuacje, w których chcemy mieć możliwość **przejścia** (ang. *crossed*) do **innego kontekstu**. Do takich sytuacji Kotlin oferuje modyfikator **crossinline**

```
1 inline fun executeSomeOperation(word: String, crossinline operation: () ->
  Unit){
2     val x = {operation()} // zagnieżdżenie
3     x()
4     println("performing operation on $word")
5 }
6
7 fun processNewWords(vararg words: String){
8     for (word in words) {
9         executeSomeOperation(word){
```

Zwróćmy uwagę, że dodanie `crossinline` pozbywa się błędu

```
Can't inline 'operation' here: it may contain non-local returns. Add 'crossinline' modifier to parameter declaration 'operation'
```

, ale dodaje nowy

```
'return' is not allowed here
```

Występują sytuacje, w których chcemy mieć możliwość **przejścia** (ang. *\*crossed\**) do **innego kontekstu**. Do takich sytuacji Kotlin oferuje modyfikator **crossinline**

```
1  inline fun executeSomeOperation(word: String, crossinline operation: () ->
    Unit){
2      val x = {operation()} // zagnieżdżenie
3      x()
4      println("performing operation on $word")
5  }
6
7  fun processNewWords(vararg words: String){
8      for (word in words) {
9          executeSomeOperation(word){
10             println("saving word")
11         }
12     }
13 }
14
15 processNewWords("Alice", "Bob", "Cherry", "Dylan")
```

# Funkcje inline

Możemy wywołać lambdy wewnątrz innej funkcji **inline**

```
1 ✓ inline fun executeSomeOperation(word: String, operation: () -> Unit){  
2     performOperation { operation() } // zagnieżdżenie  
3     println("performing operation on $word")  
4 }  
5  
6 ✓ inline fun performOperation(o: () -> Unit){  
7     o()  
8 }  
9  
10 ✓ fun processNewWords(vararg words: String){  
11 ✓     for (word in words) {  
12 ✓         executeSomeOperation(word){  
13             if (word.startsWith("C")) return  
14             println("saving word")  
15         }  
16     }  
17 }  
18  
19 processNewWords("Alice", "Bob", "Cherry", "Dylan")  
✓ [19] 135ms
```

! 2 ✓

# Funkcje anonimowe

Funkcje anonimowe w Kotlinie to funkcje, które nie mają przypisanej nazwy.

słowo  
kluczowe      lista  
parametrów      typ  
zwracany

```
1 val anonymousFunction = fun(x: Int, y: Int): Int {  
2     return x + y  
3 }  
4  
5 println(anonymousFunction(3, 5))  
✓ [43] 153ms
```

# Funkcje anonimowe

Funkcje anonimowe w Kotlinie to funkcje, które nie mają przypisanej nazwy.

```
1 val anonymousFunction = fun(x: Int, y: Int): Int {  
2     return x + y  
3 }  
4  
5 println(anonymousFunction(3, 5))  
✓ [43] 153ms
```

8

- funkcja anonimowa jest **wyrażeniem**, jak lambda,
- można stosować **return** tak samo, jak w funkcjach nazwanych,
- funkcje anonimowe **wymagają jawnej deklaracji parametrów oraz typu zwracanego** (przy użyciu ciała blokowego).

**Funkcje zakresu** (ang. **scope functions**) to jedne z najbardziej przydatnych narzędzi w Kotlinie, które pozwalają na wykonanie bloku kodu w **kontekście obiektu**.

Kotlin dostarcza pięć *scope functions*:

- **let,**
- **run,**
- **with,**
- **apply,**
- **also.**

Działają na obiekcie i tworzą dla niego **tymczasowy zakres (scope)**, w którym obiekt jest dostępny za pomocą specjalnej referencji - **this** lub **it**.

Funkcja **let** jest najczęściej używana do przetwarzania **nie-nullowalnych** obiektów oraz jako tymczasowe wprowadzenie zmiennych lokalnych. Obiekt wewnątrz bloku jest dostępny jako **it**.

```
1  val name: String? = null
2  name?.let { it: String
3      println("Witaj, $it!")
4  }
```

✓ [45] 97ms

null

Funkcja **apply** zwraca ten sam obiekt, na którym została wywołana. Dzięki temu można **łańcuchować metody** i **skonfigurować obiekt** w prosty sposób. Obiekt jest dostępny jako **this**.

```
1  class User(){
2      var name: String = ""
3      var age: Int = 0
4      var city: String = ""
5
6      override fun toString(): String {
7          return "name: $name, age: $age, city: $city"
8      }
9  }
10
11  val user = User().apply { this: User
12      name = "John"
13      age = 30
14      city = "Warszawa"
15  }
16  println(user)
✓ [48] 175ms
```

name: John, age: 30, city: Warszawa



Funkcja **run** jest podobna do **apply**, ale zwraca **wynik lambdy**, a nie obiekt. Obiekt jest dostępny jako **this**.

```
1  val userAge = User().run { this: User
2      name = "Anna"
3      age = 25
4      city = "Kraków"
5      age ^run
6  }
7  println("Wiek użytkownika: $userAge")
✓ [49] 137ms
```

Wiek użytkownika: 25

Funkcja **also** jest używana do wykonywania **dodatkowych operacji** na obiekcie, **bez zmiany jego właściwości**. W bloku kodu obiekt jest dostępny jako **it**.

```
1 val numbers = mutableListOf("jeden", "dwa", "trzy")
2 numbers.also { it: MutableList<String>
3     println("Lista przed modyfikacją: $it")
4 }.add("cztery")
5 println(numbers)
✓ [50] 151ms
```

```
Lista przed modyfikacją: [jeden, dwa, trzy]
[jeden, dwa, trzy, cztery]
```

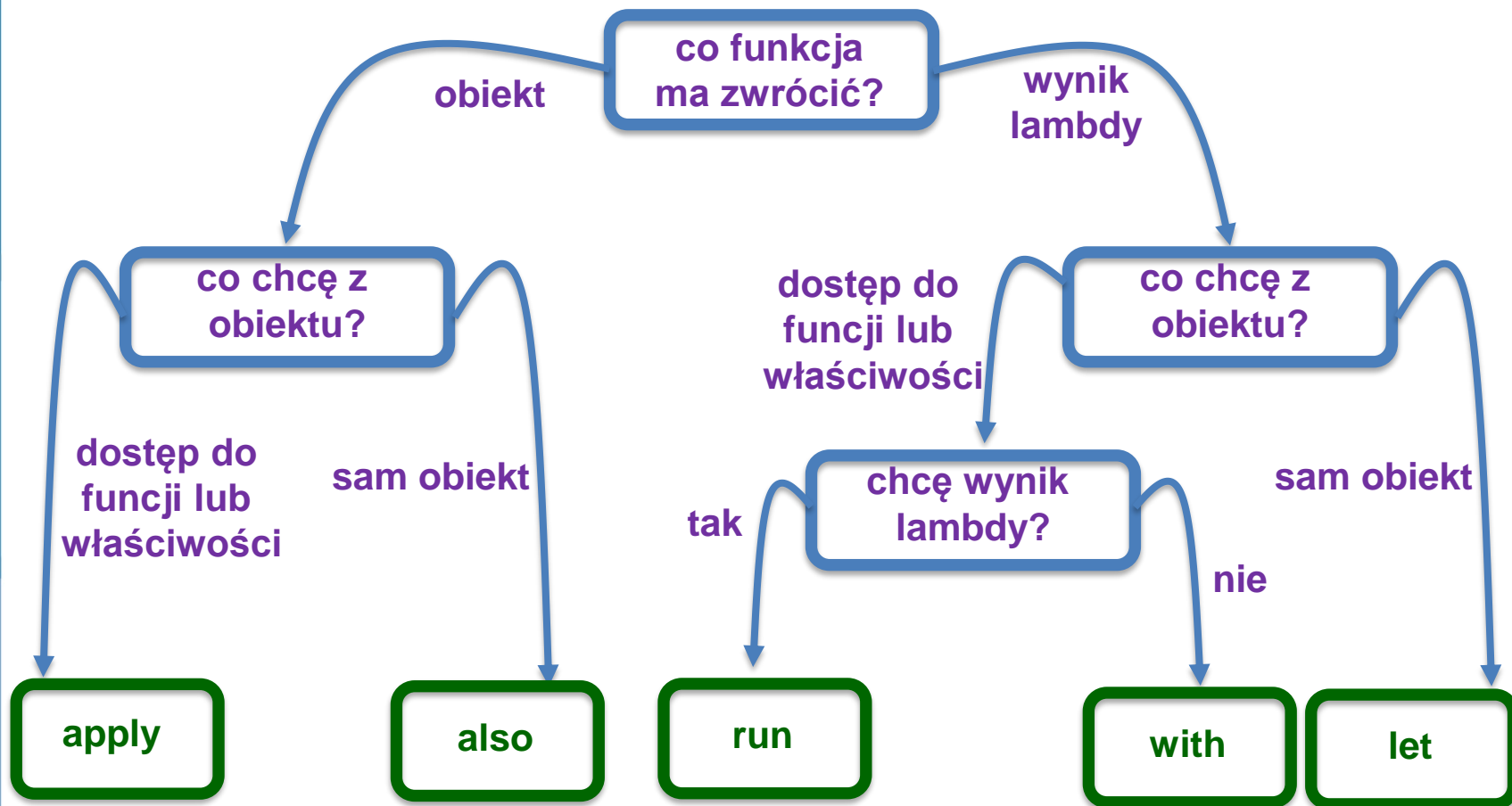
Funkcja **with** jest wywoływana z obiektem przekazywanym jako argument. Jest przydatna, gdy chcemy wykonać kilka operacji na obiekcie i nie potrzebujemy wyniku zwróconego przez funkcję.

```
1  val formattedText = with(StringBuilder()) { this: StringBuilder
2      append("Kotlin Scope Functions\n")
3      append("=====\n")
4      append("1. let\n")
5      append("2. run\n")
6      append("3. with\n")
7      append("4. apply\n")
8      append("5. also\n")
9      toString() ^with // Zwracamy wynik operacji jako string
10 }
11
12 println(formattedText)
✓ [52] 143ms
```

```
Kotlin Scope Functions
=====
1. let
2. run
3. with
4. apply
5. also
```

# Funkcje zakresu

	this	it
Zwraca wynik lambdy	run	let
Zwraca obiekt	apply	also



**Przeciążanie operatorów** polega na zaimplementowaniu funkcji o określonej nazwie, oznaczonej modyfikatorem **operator**. Funkcja może być metodą klasy lub funkcją rozszerzającą.

```
1  data class Point(val x: Int, val y: Int)
2
3  // Przeciążenie operatora unaryMinus
4  operator fun Point.unaryMinus() = Point(-x, -y)
5
6  val point = Point(10, 20)
7  println(-point) // Wynik: Point(x=-10, y=-20)
✓ [54] 103ms
    Point(x=-10, y=-20)
```

Expression	Translated to
<code>+a</code>	<code>a.unaryPlus()</code>
<code>-a</code>	<code>a.unaryMinus()</code>
<code>!a</code>	<code>a.not()</code>

Expression	Translated to
<code>a++</code>	<code>a.inc()</code>
<code>a--</code>	<code>a.dec()</code>
<code>!a</code>	<code>a.not()</code>

Expression	Translated to
<code>a + b</code>	<code>a.plus(b)</code>
<code>a - b</code>	<code>a.minus(b)</code>
<code>a * b</code>	<code>a.times(b)</code>
<code>a / b</code>	<code>a.div(b)</code>
<code>a % b</code>	<code>a.rem(b)</code>
<code>a..b</code>	<code>a.rangeTo(b)</code>
<code>a..&lt;b</code>	<code>a.rangeUntil(b)</code>