Uniwersytet Wrocławski

Rafał Lewandków

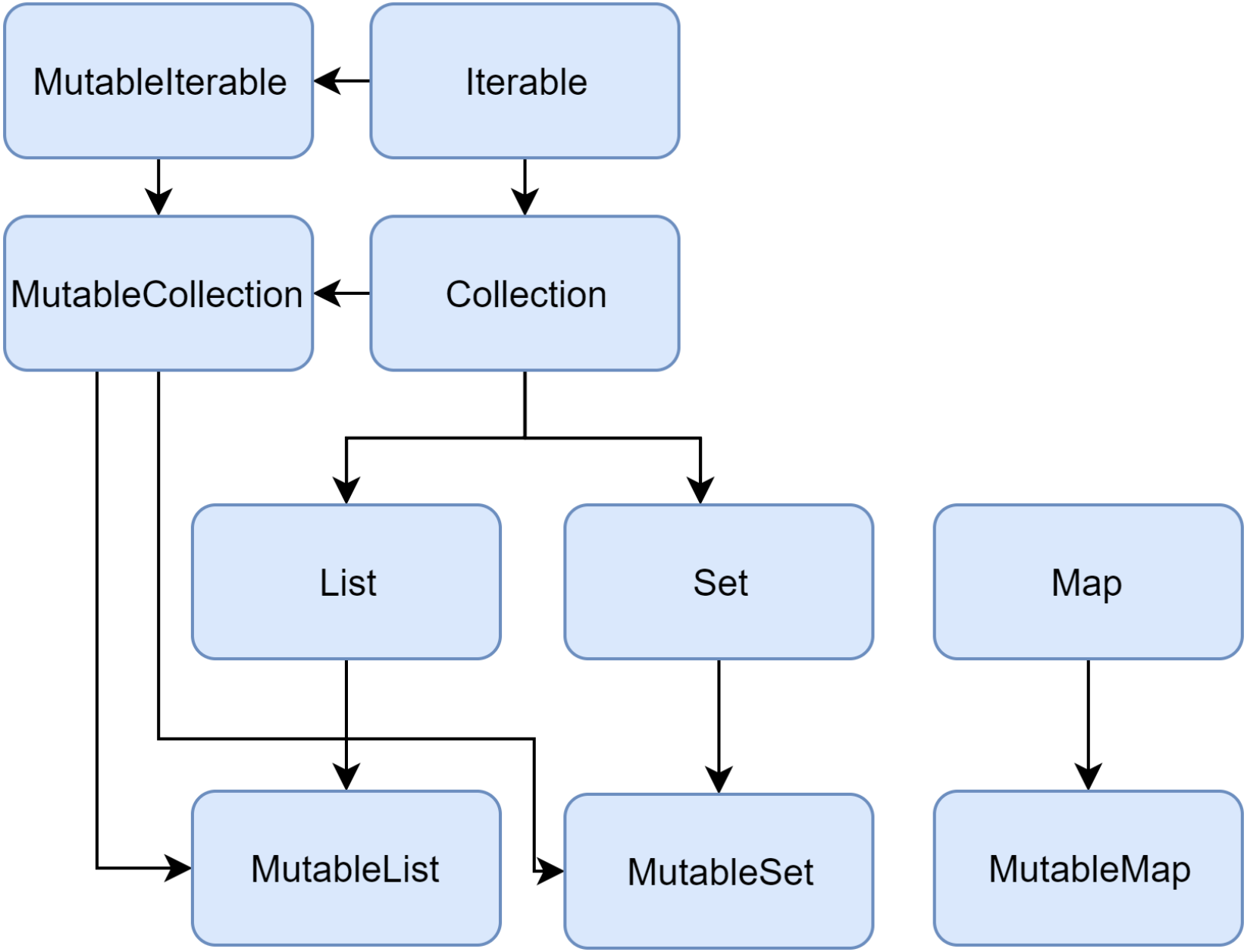
PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 1

WYKŁAD 4

- Kolekcje

Uniwersytet Wrocławski

Kolekcje



Uniwersytet Wrocławski

List<T>, MutableList<T>

```
1  val numbers = listOf("one", "two", "three", "four")
2
3  println("Liczba elementów: ${numbers.size}")
4  println("Trzeci element: ${numbers.get(2)}") // lub numbers[2]
5  println("Czwarty element: ${numbers[3]}")
6  println("Indeks elementu \"two\": ${numbers.indexOf("two")}")
   [20]
```

```
Liczba elementów: 4
Trzeci element: three
Czwarty element: four
Indeks elementu "two": 1
```

Uniwersytet Wrocławski

List<T>, MutableList<T>

```
1  val numbers = listOf("one", "two", "three", "four")
2
3  println("Liczba elementów: ${numbers.size}")
4  println("Trzeci element: ${numbers.get(2)}") // lub numbers[2]
5  println("Czwarty element: ${numbers[3]}")
6  println("Indeks elementu \"two\": ${numbers.indexOf("two")}")
   [20]
```

```
Liczba elementów: 4
Trzeci element: three
Czwarty element: four
Indeks elementu "two": 1
```

- Elementy przechowywane **w określonej kolejności**
- Dostęp za pomocą **indeksów**
- Elementy **mogą się powtarzać**
- Dwie listy są równe, jeśli mają tę samą wielkość i takie same elementy na tych samych pozycjach

Uniwersytet Wrocławski

List<T>, MutableList<T>

MutableList<T> rozszerza funkcjonalność **List<T>** o operacje zapisu, takie jak dodawanie, usuwanie i modyfikowanie elementów.

```
1  val numbers = mutableListOf(1, 2, 3, 4)
2
3  numbers.add(5)            // Dodanie elementu
4  numbers.removeAt(1)       // Usunięcie elementu z indeksu 1
5  numbers[0] = 0            // Zmiana wartości elementu na indeksie 0
6  numbers.shuffle()         // Przestawienie elementów w losowej kolejności
7
8  println(numbers)
   [22]
```

```
[3, 4, 5, 0]
```

Uniwersytet Wrocławski

Set<T> i MutableSet<T>

```
1  val numbers = setOf(1, 2, 3, 4)
2
3  println("Liczba elementów: ${numbers.size}")
4  if (numbers.contains(1)) println("1 znajduje się w zbiorze")
5
6  val numbersBackwards = setOf(4, 3, 2, 1)
7  println("Zbiory są równe: ${numbers == numbersBackwards}")
   [23]
```

```
Liczba elementów: 4
1 znajduje się w zbiorze
Zbiory są równe: true
```

Uniwersytet Wrocławski

Set<T> i MutableSet<T>

```
1  val n1 = mutableSetOf(1, 2, 3, 4)
2  val n2 = setOf(1, 2, 3, 4)
3  println(n1::class.simpleName)
4  println(n2::class.simpleName)
   [26]
```

```
LinkedHashSet
LinkedHashSet
```

```
1  val numbers = mutableSetOf(1, 2, 3, 4)
2  numbers.add(5)
3  numbers.add(5)
4  numbers.add(5)
5  println(numbers)
   [65]
```

```
[1, 2, 3, 4, 5]
```

- Kolejność elementów w zbiorach domyślnie **nie jest zdefiniowana.**
- W Kotlinie elementy przechowywane w określonej kolejności – **LinkedHashSet<T>**
- Przechowuje **unikalne elementy**
- Dwa zbiory są równe, jeśli mają taką samą liczbę elementów i każdy element jednego zbioru jest równy jednemu elementowi w drugim zbiorze

Uniwersytet Wrocławski

Set<T> i MutableSet<T>

MutableSet<T> rozszerza funkcjonalność **Set<T>** o operacje zapisu, takie jak dodawanie czy usuwanie elementów.

```
1  val numbers = mutableSetOf(1, 2, 3, 4)
2
3  numbers.add(5)          // Dodanie elementu
4  numbers.remove(2)       // Usunięcie elementu
5  println(numbers)        // Wynik: [1, 3, 4, 5]
   [24]
```

```
[1, 3, 4, 5]
```

Uniwersytet Wrocławski

Map<K, V> i MutableMap<K, V>

```
1  val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
2
3  println("Wszystkie klucze: ${numbersMap.keys}")
4  println("Wszystkie wartości: ${numbersMap.values}")
5
6  println(numbersMap["key1"])
   [51]
```

```
Wszystkie klucze: [key1, key2, key3, key4]
Wszystkie wartości: [1, 2, 3, 1]
1
```

Uniwersytet Wrocławski

Map<K, V> i MutableMap<K, V>

```
1  val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3, "key4" to 1)
2
3  println("Wszystkie klucze: ${numbersMap.keys}")
4  println("Wszystkie wartości: ${numbersMap.values}")
5
6  println(numbersMap["key1"])
   [51]
```

```
Wszystkie klucze: [key1, key2, key3, key4]
Wszystkie wartości: [1, 2, 3, 1]
1
```

- Kolejność elementów w mapach domyślnie **nie jest zdefiniowana.**
- W Kotlinie elementy przechowywane w określonej kolejności – **LinkedHashMap<K, V>**
- Przechowuje **pary *klucz-wartość***
- Dwie mapy są równe, jeśli zawierają te same pary klucz-wartość, niezależnie od kolejności tych par.

Uniwersytet Wrocławski

Map<K, V> i MutableMap<K, V>

```
1  val m1 = mutableMapOf("one" to 1, "two" to 2)
2  val m2 = mapOf("one" to 1, "two" to 2)
3  println(m1::class.simpleName)
4  println(m2::class.simpleName)
   [55]
```

```
LinkedHashMap
LinkedHashMap
```

- Kolejność elementów w mapach domyślnie **nie jest zdefiniowana.**
- W Kotlinie elementy przechowywane w określonej kolejności – **LinkedHashMap<K, V>**
- Przechowuje **pary *klucz-wartość***
- Dwie mapy są równe, jeśli zawierają te same pary klucz-wartość, niezależnie od kolejności tych par.

Uniwersytet Wrocławski

Map<K, V> i MutableMap<K, V>

MutableMap<K,V> rozszerza funkcjonalność **Map<K,V>** o operacje zapisu, takie jak dodawanie czy usuwanie elementów.

```
1  val numbersMap = mutableMapOf("one" to 1, "two" to 2)
2
3  numbersMap.put("three", 3)      // Dodanie nowej pary
4  numbersMap["one"] = 11          // Aktualizacja wartości dla klucza "one"
5
6  println(numbersMap)
   [54]
```

```
{one=11, two=2, three=3}
```

Map<K, V> i MutableMap<K, V>

```
for (key in map.keys) {
    println(key)
    println(map[key])
}
```

```
[61]
```

```
for (value in map.values) {
    println(value)
}
```

```
[63]
```

```
for (entry in map.entries.iterator()) {
    println("${entry.key} : ${entry.value}")
}
```

Uniwersytet Wrocławski

Wybrane operacje na kolekcjach

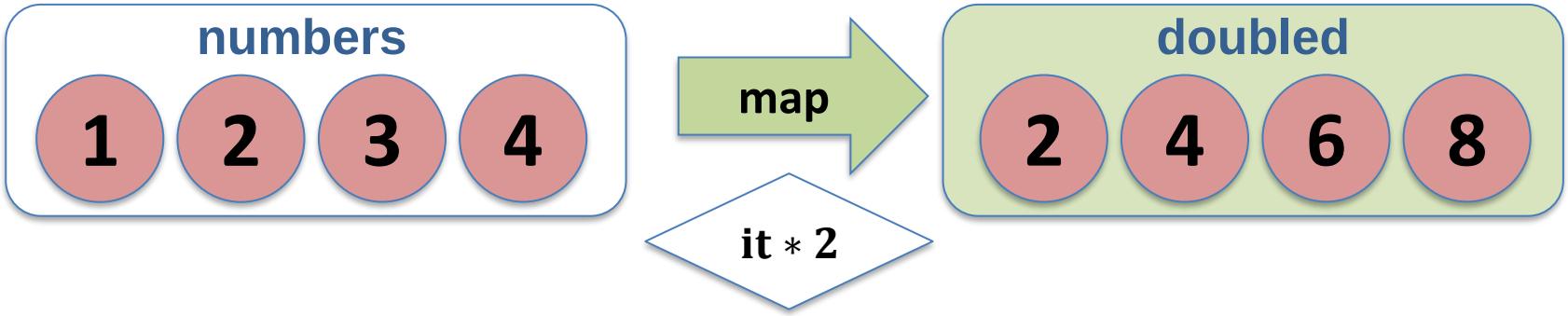
Transformacje

Mapowanie

Transformacje mapujące to narzędzie, które pozwala na przekształcenie kolekcji na podstawie wyników funkcji zastosowanej do jej elementów.

```
1  val numbers = listOf(1, 2, 3, 4)
2  val doubled = numbers.map { it * 2 }
3  println(numbers)
4  println(doubled) // Wynik: [2, 4, 6, 8]
   [2]
```

```
[1, 2, 3, 4]
[2, 4, 6, 8]
```



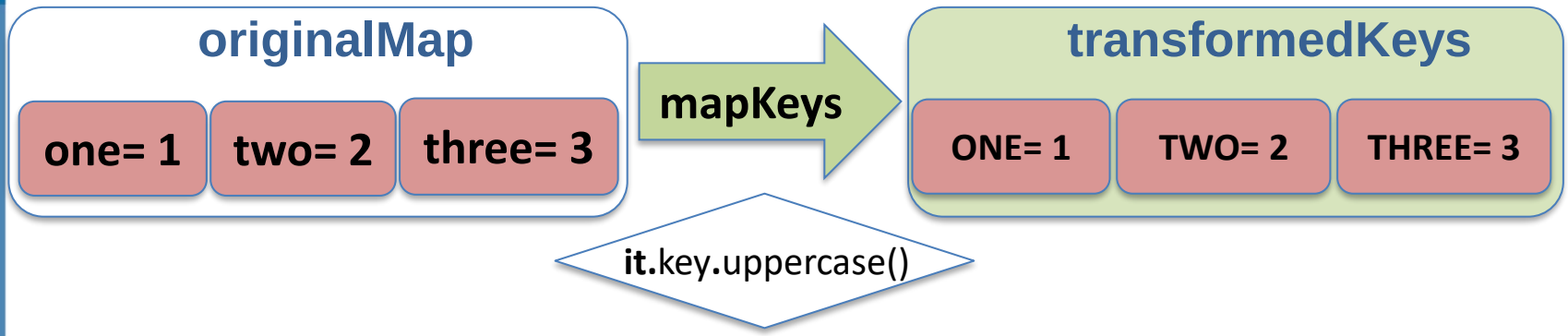
Transformacje

Mapowanie

W przypadku pracy z mapami (**Map<K, V>**) Kotlin umożliwia transformację kluczy lub wartości bez konieczności zmieniania całej mapy.

```
1  val originalMap = mapOf("one" to 1, "two" to 2, "three" to 3)
2  println(originalMap)
3  val transformedKeys = originalMap.mapKeys { it.key.uppercase() }
4  println(transformedKeys) // Wynik: {ONE=1, TWO=2, THREE=3}
   ✓ [1] 680ms
```

```
{one=1, two=2, three=3}
{ONE=1, TWO=2, THREE=3}
```



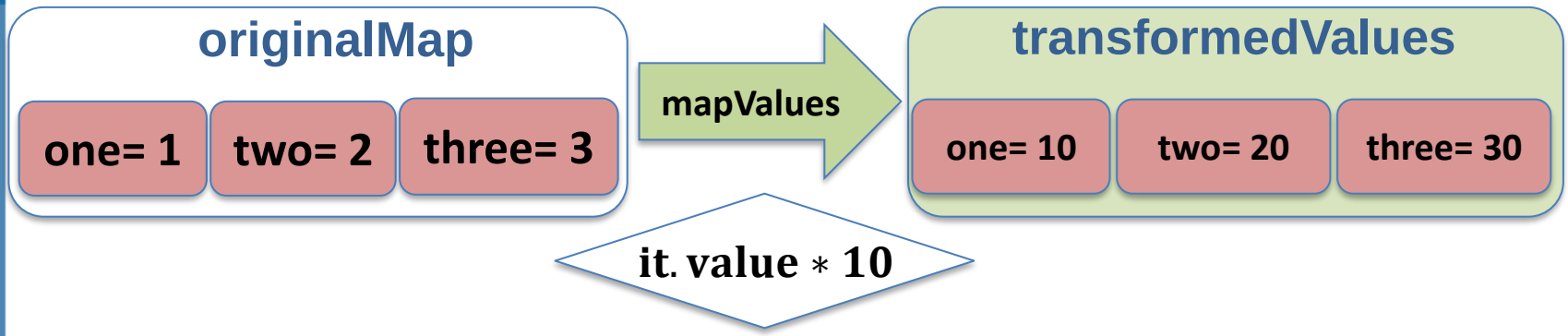
Transformacje

Mapowanie

W przypadku pracy z mapami (**Map<K, V>**) Kotlin umożliwia transformację kluczy lub wartości bez konieczności zmieniania całej mapy.

```
1  val originalMap = mapOf("one" to 1, "two" to 2, "three" to 3)
2  println(originalMap)
3  val transformedValues = originalMap.mapValues { it.value * 10 }
4  println(transformedValues) // Wynik: {one=10, two=20, three=30}
   ✓ [2] 862ms
```

```
{one=1, two=2, three=3}
{one=10, two=20, three=30}
```



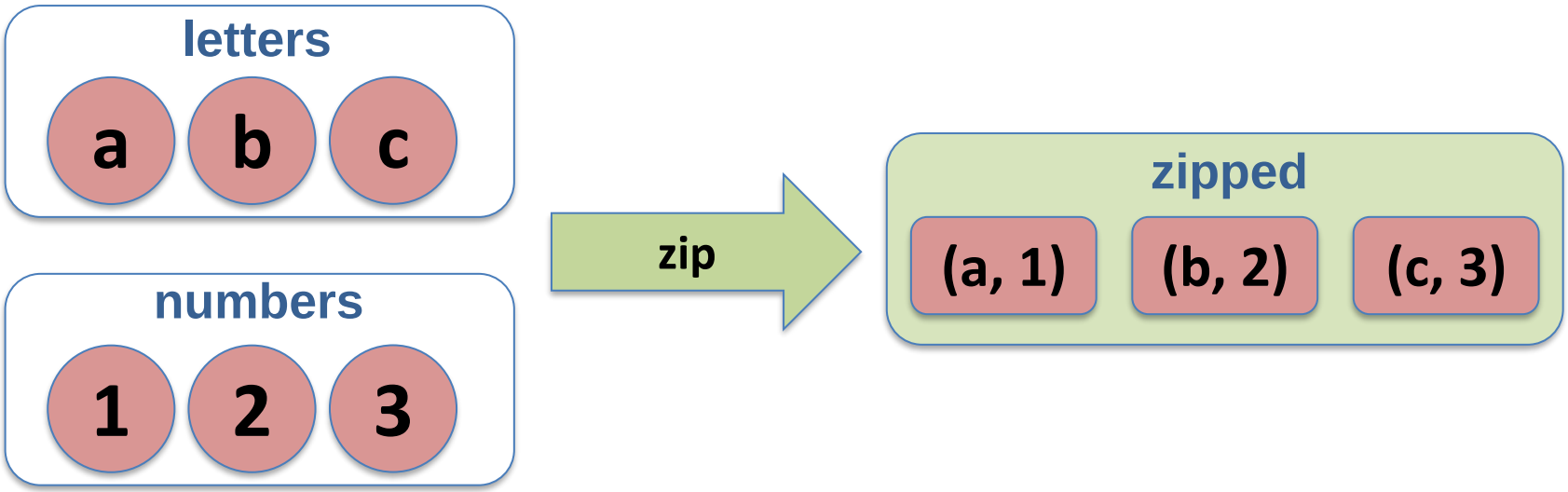
Transformacje

Zip

Transformacja pozwala tworzyć pary na podstawie elementów o tych samych pozycjach w dwóch kolekcjach.

```
1  val letters = listOf("a", "b", "c")
2  val numbers = listOf(1, 2, 3)
3  val zipped = letters.zip(numbers)
4  println(zipped)
   [7]
```

```
[(a, 1), (b, 2), (c, 3)]
```



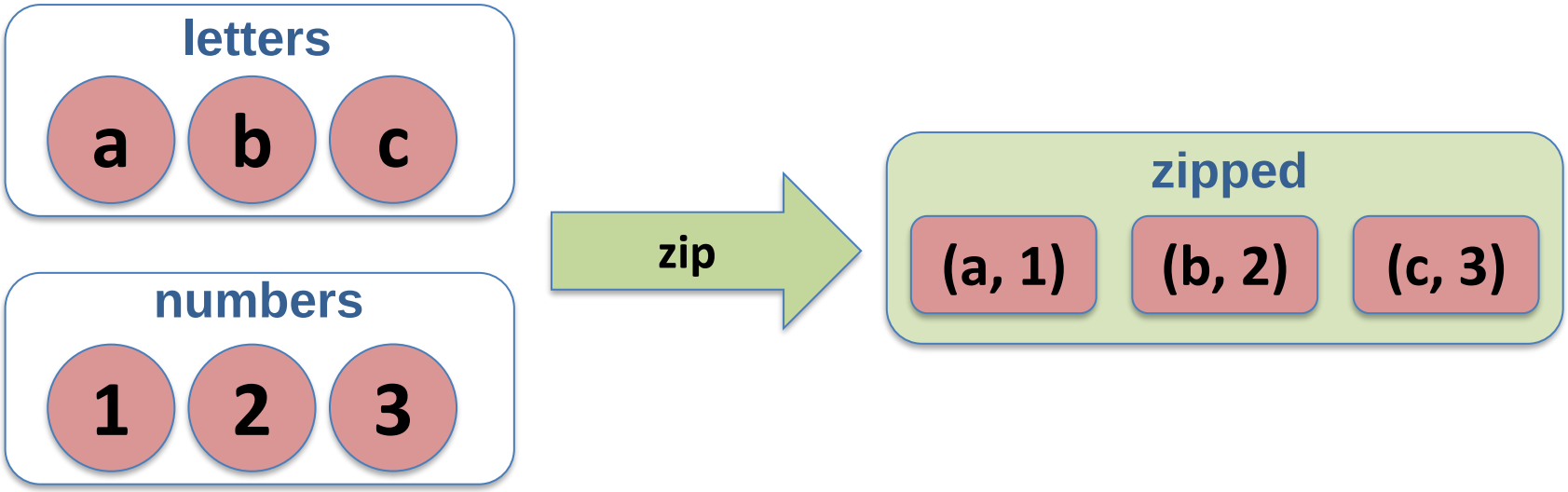
Transformacje

Zip

Transformacja pozwala tworzyć pary na podstawie elementów o tych samych pozycjach w dwóch kolekcjach.

```
1  val letters = listOf("a", "b", "c")
2  val numbers = listOf(1, 2, 3)
3  val zipped = letters zip numbers
4  println(zipped) // Wynik: [(a, 1), (b, 2), (c, 3)]
   [9]
```

```
[(a, 1), (b, 2), (c, 3)]
```

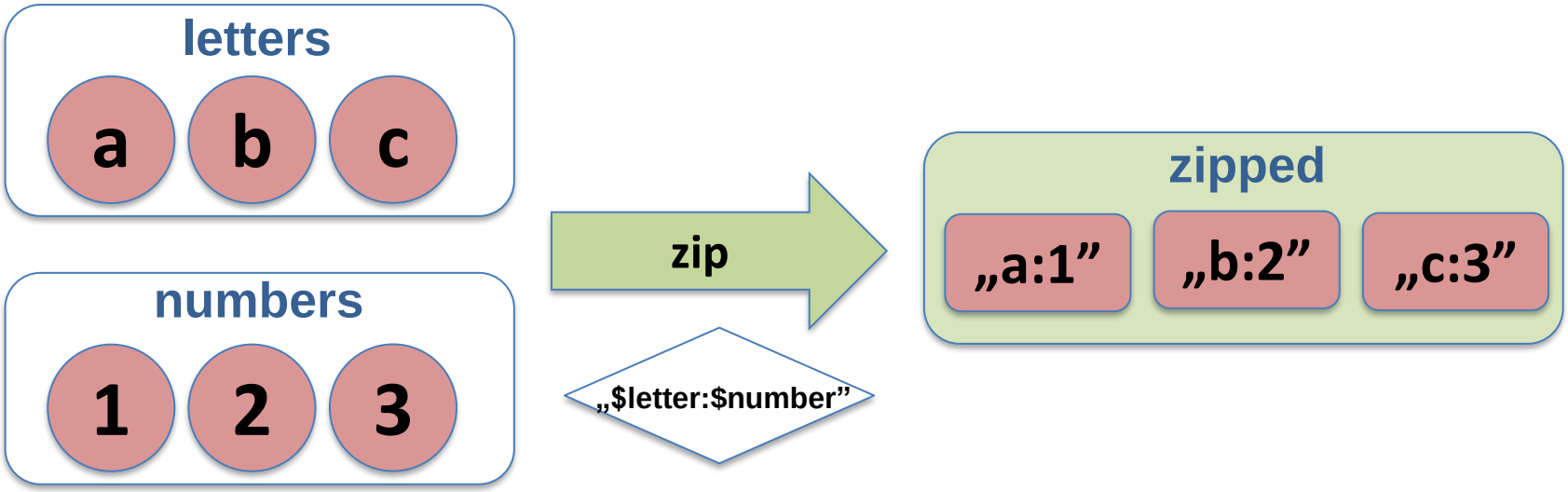


Transformacje

Zip

```
1  val letters = listOf("a", "b", "c")
2  val numbers = listOf(1, 2, 3)
3  val transformed = letters.zip(numbers) { letter, number ->
    "$letter:$number" }
4  println(transformed) // Wynik: [a:1, b:2, c:3]
   [10]
```

```
[a:1, b:2, c:3]
```



Transformacje

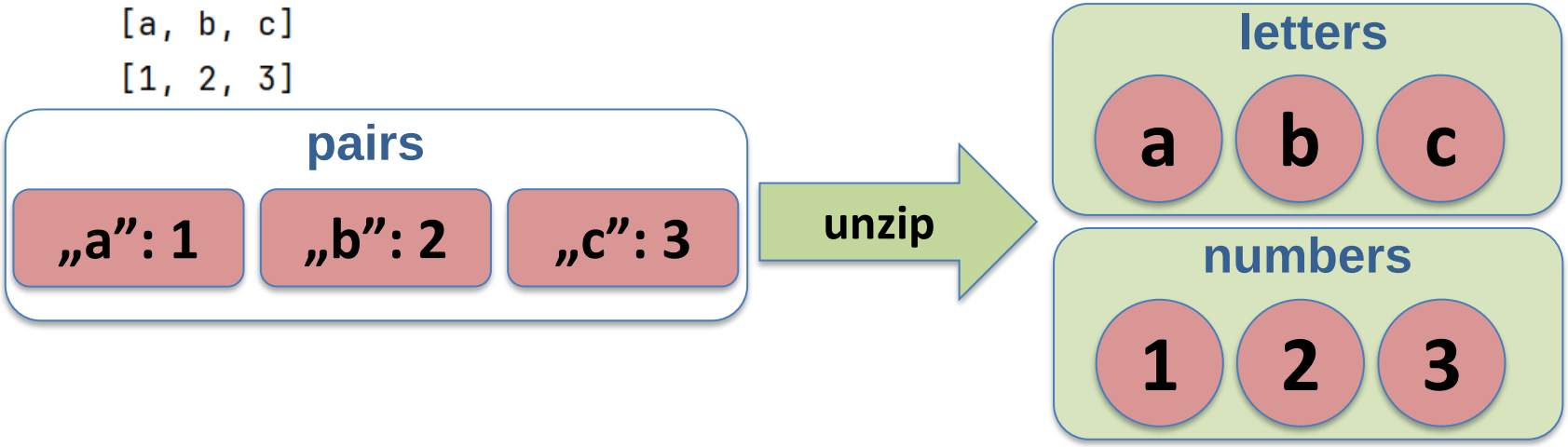
Unzip

Funkcja pozwala na wykonanie odwrotnej operacji – dzieli listę par na dwie osobne listy:

- Pierwsza lista zawiera pierwsze elementy każdej pary.
- Druga lista zawiera drugie elementy każdej pary.

```
1  val pairs = listOf("a" to 1, "b" to 2, "c" to 3)
2  val (letters, numbers) = pairs.unzip() // deklaracja destrukturyzująca
3  println(letters) // Wynik: [a, b, c]
4  println(numbers) // Wynik: [1, 2, 3]
   [11]
```

```
[a, b, c]
[1, 2, 3]
```



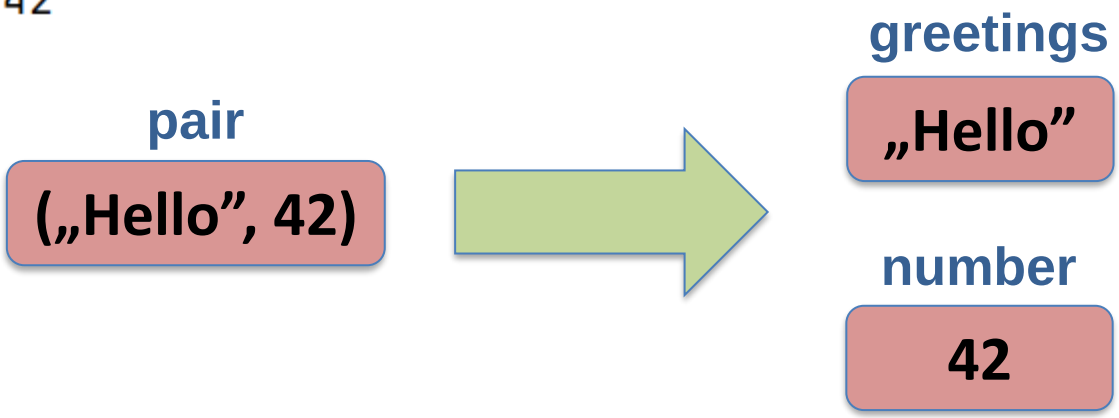
Uniwersytet Wrocławski

Deklaracja Destrukturyzująca

Deklaracje destrukturyzujące to funkcja, która umożliwia rozbicie obiektu na jego składowe w jednym kroku. Jest to szczególnie przydatne podczas pracy z kolekcjami, parami (**Pair**), trójkami (**Triple**).

```
1  val pair = "Hello" to 42
2  val (greeting, number) = pair
3  println(greeting) // Wynik: Hello
4  println(number)   // Wynik: 42
   ✓ [3] 247ms
```

```
Hello
42
```

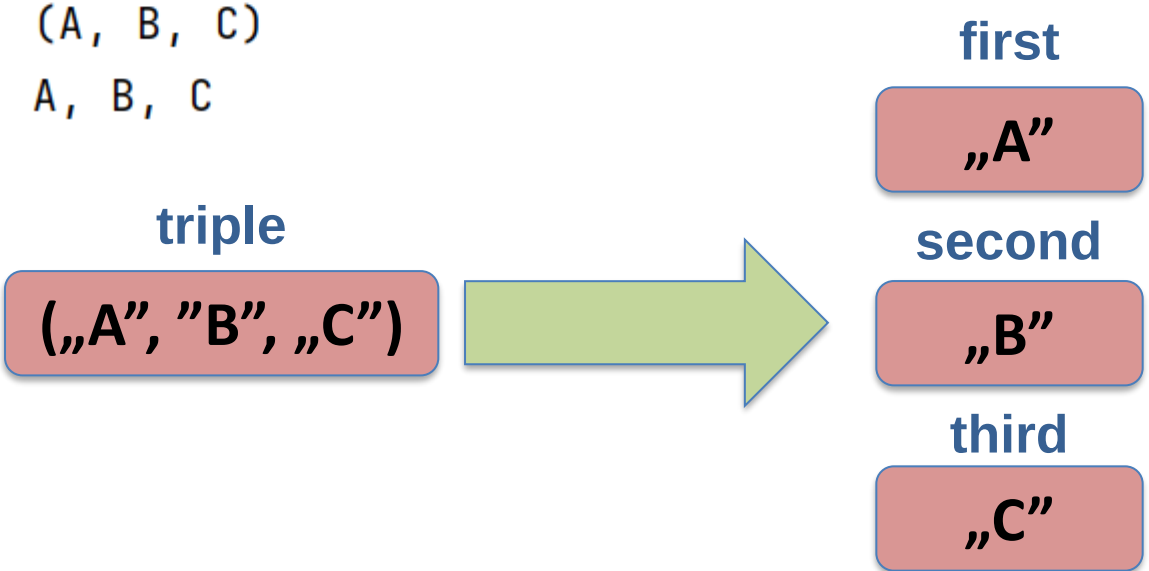


Deklaracja Destrukturyzująca

Deklaracje destrukturyzujące to funkcja, która umożliwia rozbicie obiektu na jego składowe w jednym kroku. Jest to szczególnie przydatne podczas pracy z kolekcjami, parami (**Pair**), trójkami (**Triple**).

```
1  val triple = Triple("A", "B", "C")
2  val (first, second, third) = triple
3  println(triple)
4  println("$first, $second, $third")
   ✓ [6] 192ms
```

```
(A, B, C)
A, B, C
```



Transformacje

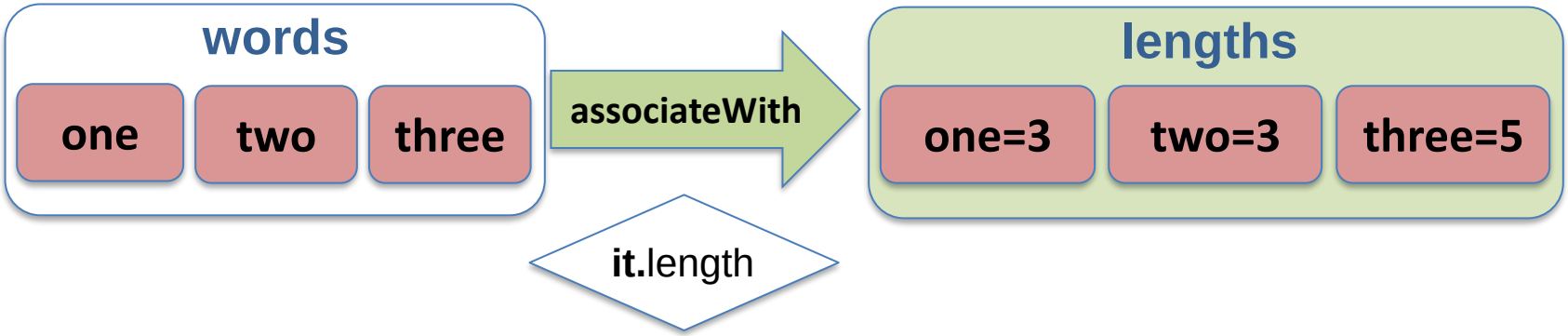
Asocjacja

Transformacje asocjacyjne umożliwiają budowanie map (**Map**) na podstawie elementów kolekcji oraz wartości z nimi powiązanych.

Funkcja **associateWith()** tworzy mapę, gdzie elementy oryginalnej kolekcji stają się kluczami.

```
1  val words = listOf("one", "two", "three")
2  val lengths = words.associateWith { it.length }
3  println(lengths) // Wynik: {one=3, two=3, three=5}
   [14]
```

```
{one=3, two=3, three=5}
```



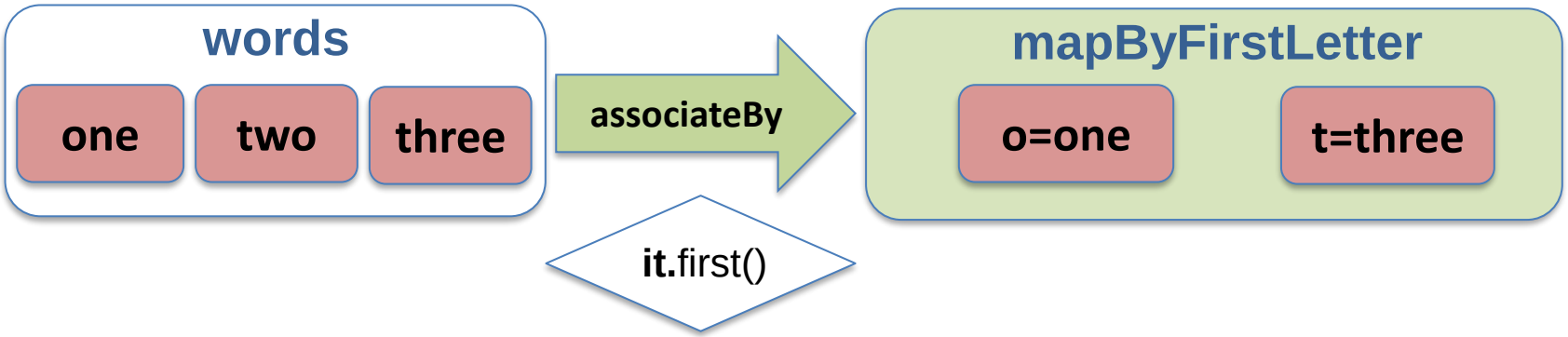
Transformacje

Asocjacja

Funkcja **associateBy**() buduje mapę, w której elementy kolekcji są wartościami, a klucze są generowane na podstawie podanej funkcji. W przypadku kluczy-duplikatów, w mapie zostaje tylko ostatnia para.

```
1  val words = listOf("one", "two", "three")
2  val mapByFirstLetter = words.associateBy { it.first() }
3  println(mapByFirstLetter) // Wynik: {o=one, t=three}
   [15]
```

```
{o=one, t=three}
```



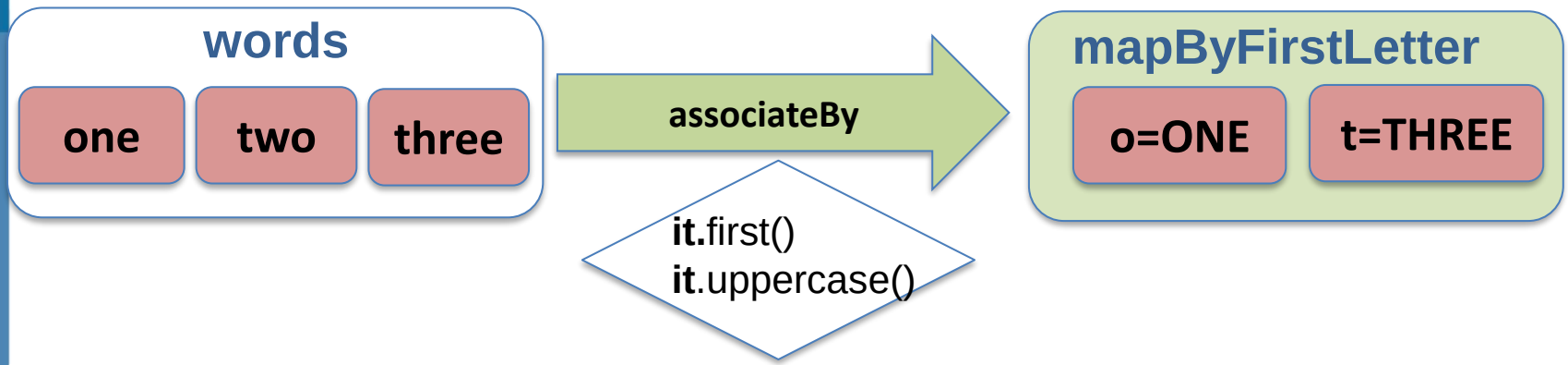
Transformacje

Asocjacja

Można dodać drugą funkcję transformującą, która określa, jak przekształcić wartości w mapie.

```
1  val words = listOf("one", "two", "three")
2  val mapByFirstLetter = words.associateBy(
3      keySelector = { it.first() },
4      valueTransform = { it.uppercase() }
5  )
6  println(mapByFirstLetter) // Wynik: {o=ONE, t=THREE}
   [16]
```

```
{o=ONE, t=THREE}
```



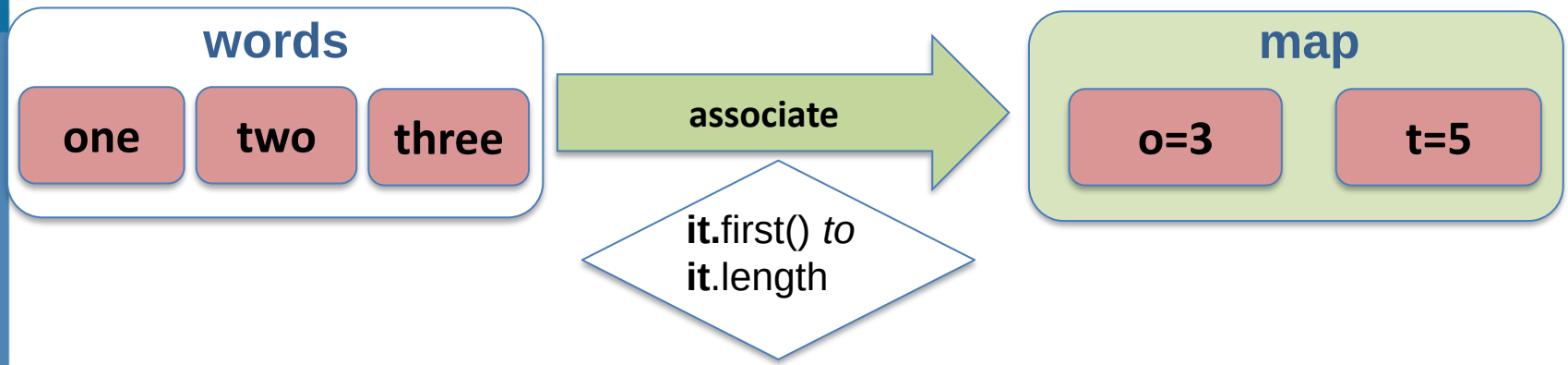
Transformacje

Asocjacja

Funkcja **associate()** pozwala zbudować mapę, w której zarówno klucz, jak i wartość są tworzone na podstawie elementu kolekcji za pomocą funkcji zwracającej parę (**Pair**).

```
1  val words = listOf("one", "two", "three")
2  val map = words.associate { it.first() to it.length }
3  println(map) // Wynik: {o=3, t=5}
   [17]
```

```
{o=3, t=5}
```



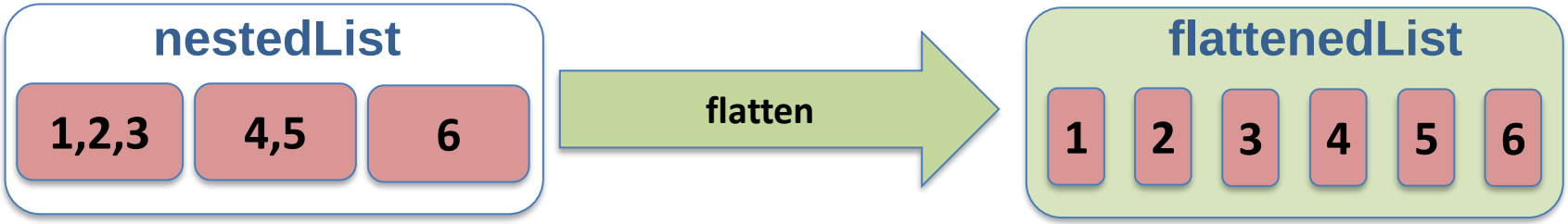
Transformacje

Spłaszczanie kolekcji

Funkcja **flatten()** działa na kolekcji kolekcji, np. **List<Set<T>>** lub **List<List<T>>**. Wynikiem jest pojedyncza lista zawierająca wszystkie elementy z kolekcji zagnieżdżonych.

```
1  val nestedList = listOf(listOf(1, 2, 3), listOf(4, 5), listOf(6))
2  println(nestedList)
3  val flattenedList = nestedList.flatten()
4  println(flattenedList) // Wynik: [1, 2, 3, 4, 5, 6]
   [20]
```

```
[[1, 2, 3], [4, 5], [6]]
[1, 2, 3, 4, 5, 6]
```



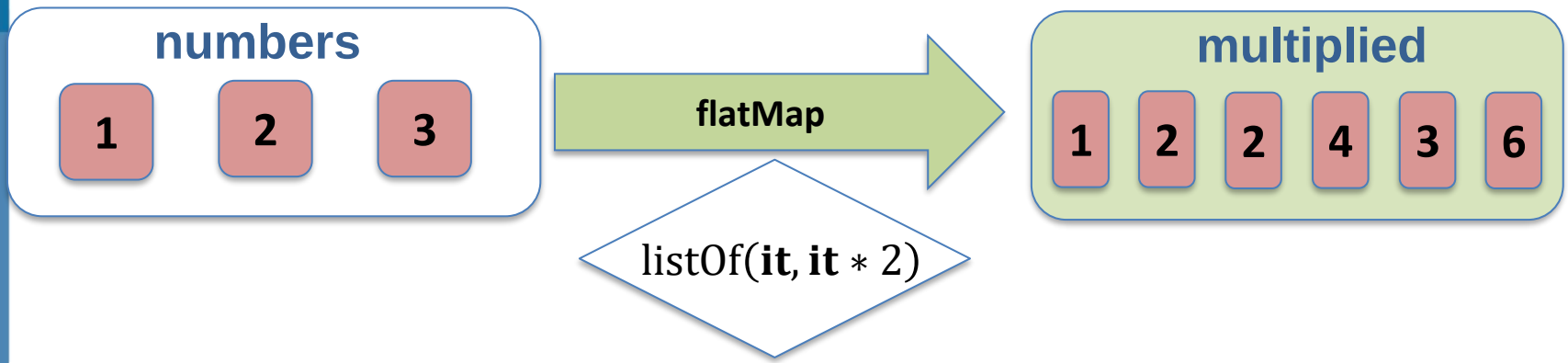
Transformacje

Spłaszczanie kolekcji

Funkcja **flatMap()** jest bardziej uniwersalna. Działa podobnie jak kombinacja **map()** i **flatten()**. Najpierw mapuje elementy kolekcji wejściowej na inne kolekcje, a następnie łączy wyniki w jedną listę.

```
1  val numbers = listOf(1, 2, 3)
2  val multiplied = numbers.flatMap { listOf(it, it * 2) }
3  println(multiplied) // Wynik: [1, 2, 2, 4, 3, 6]
   ✓ [8] 178ms
```

```
[1, 2, 2, 4, 3, 6]
```



Filtrowanie

Przez predykat

W Kotlinie warunki filtracji definiowane są za pomocą **predykatów** – funkcji lambda, które przyjmują element kolekcji i zwracają wartość logiczną:

- **true** oznacza, że dany element spełnia warunek predykatu,
- **false** oznacza, że element nie spełnia warunku.

Te funkcje **nie modyfikują oryginalnej kolekcji**, dlatego można ich używać zarówno na kolekcjach mutowalnych, jak i tylko do odczytu.

Filtrowanie

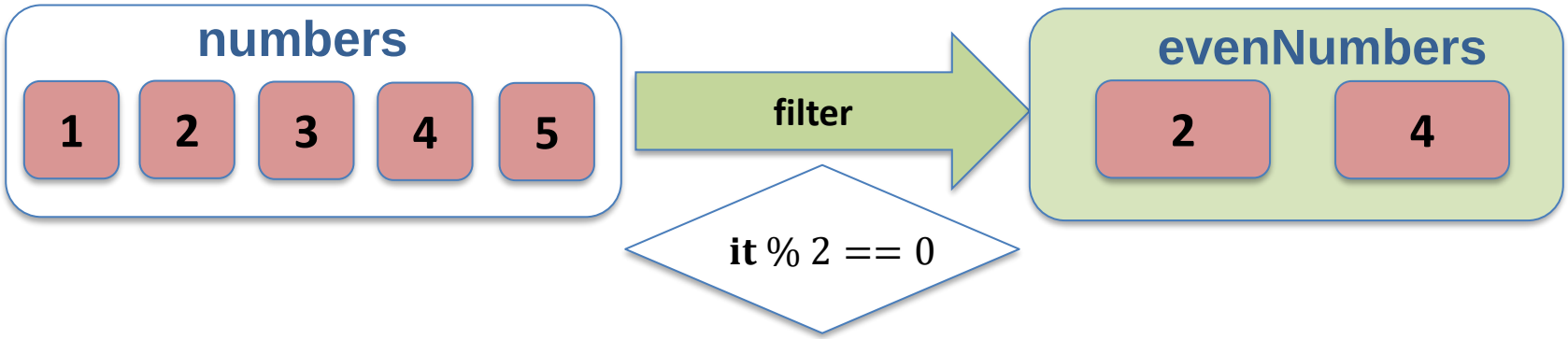
Przez predykat

Funkcja **filter()** zwraca elementy kolekcji, które spełniają podany predykat. Działa zarówno na listach, zbiorach, jak i mapach. W przypadku map filtracja odbywa się na wartościach.

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

[1]

```
[2, 4]
```



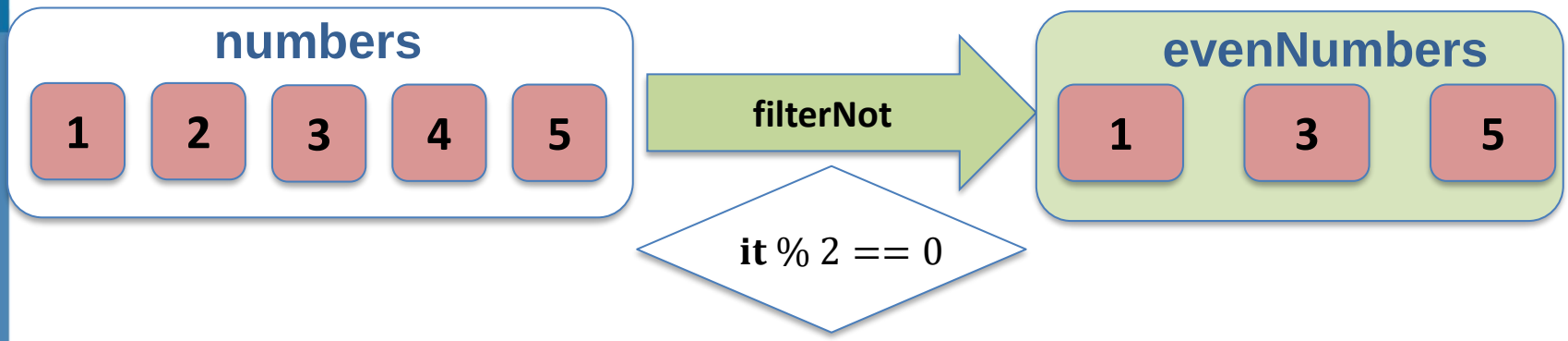
Filtrowanie

Przez predykat

Funkcja **filterNot()** zwraca elementy kolekcji, które nie spełniają podanego predykatu.

```
1  val numbers = listOf(1, 2, 3, 4, 5)
2  val oddNumbers = numbers.filterNot { it % 2 == 0 } // Filtruj liczby
    nieparzyste
3  println(oddNumbers) // Wynik: [1, 3, 5]
   [4]
```

```
[1, 3, 5]
```



Filtrowanie

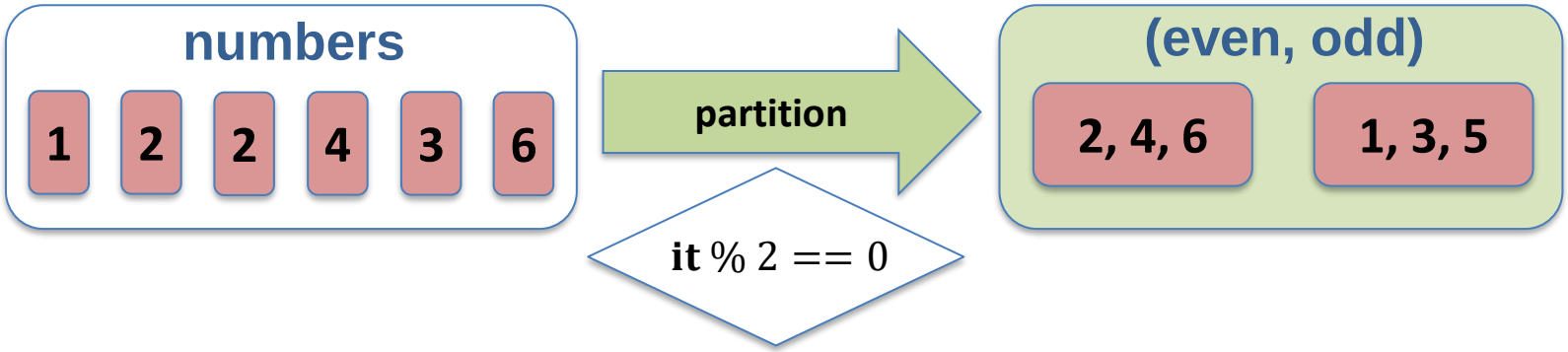
Partition

Funkcja **partition()** pozwala na jednoczesne rozdzielenie elementów kolekcji na dwie grupy:

- elementy spełniające podany predykat,
- elementy, które nie spełniają tego warunku.

```
1  val numbers = listOf(1, 2, 3, 4, 5, 6)
2  val (even, odd) = numbers.partition { it % 2 == 0 }
3
4  println("Liczby parzyste: $even")   // Wynik: [2, 4, 6]
5  println("Liczby nieparzyste: $odd") // Wynik: [1, 3, 5]
   [5]
```

```
Liczby parzyste: [2, 4, 6]
Liczby nieparzyste: [1, 3, 5]
```



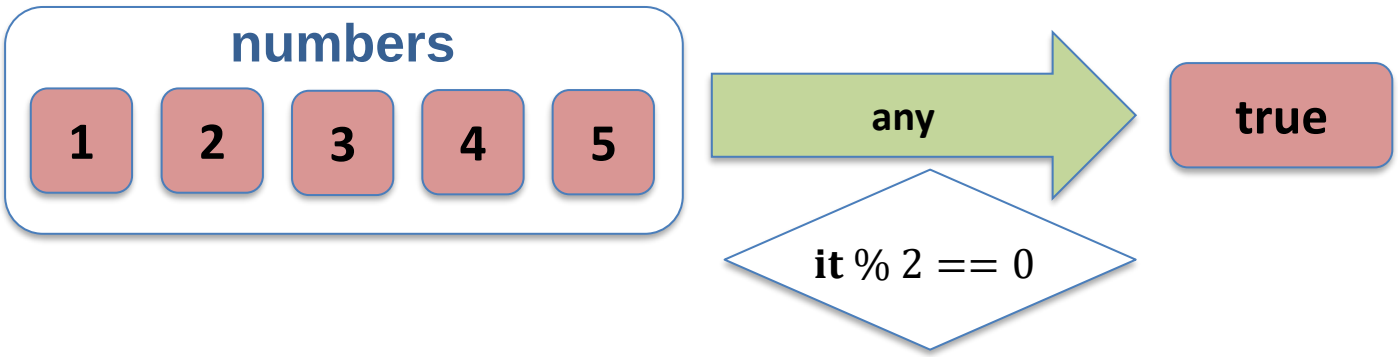
Filtrowanie

Testowanie predykatów

Kotlin oferuje funkcje, które umożliwiają sprawdzanie, czy elementy kolekcji spełniają określony warunek (predykat).

```
1  val numbers = listOf(1, 2, 3, 4, 5)
2  val hasEven = numbers.any { it % 2 == 0 }
3
4  println("Czy kolekcja zawiera liczby parzyste? $hasEven") // Wynik: true
   [6]
```

```
Czy kolekcja zawiera liczby parzyste? true
```



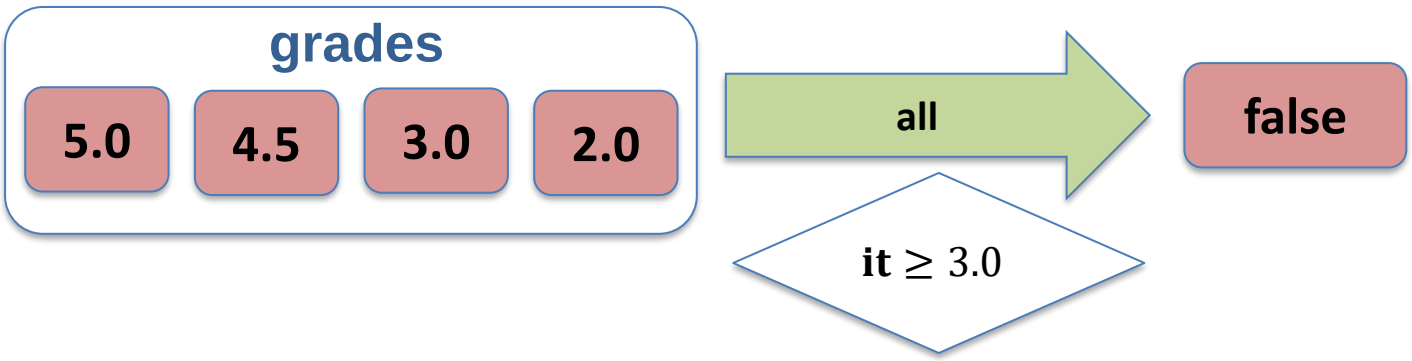
Filtrowanie

Testowanie predykatów

Kotlin oferuje funkcje, które umożliwiają sprawdzanie, czy elementy kolekcji spełniają określony warunek (predykat).

```
1  val grades = listOf(5.0, 4.5, 3.0, 2.0)
2  val allPassed = grades.all { it >= 3.0 }
3
4  println("Czy wszyscy zdali egzamin? $allPassed")
   [10]
```

```
Czy wszyscy zdali egzamin? false
```



Grupowanie

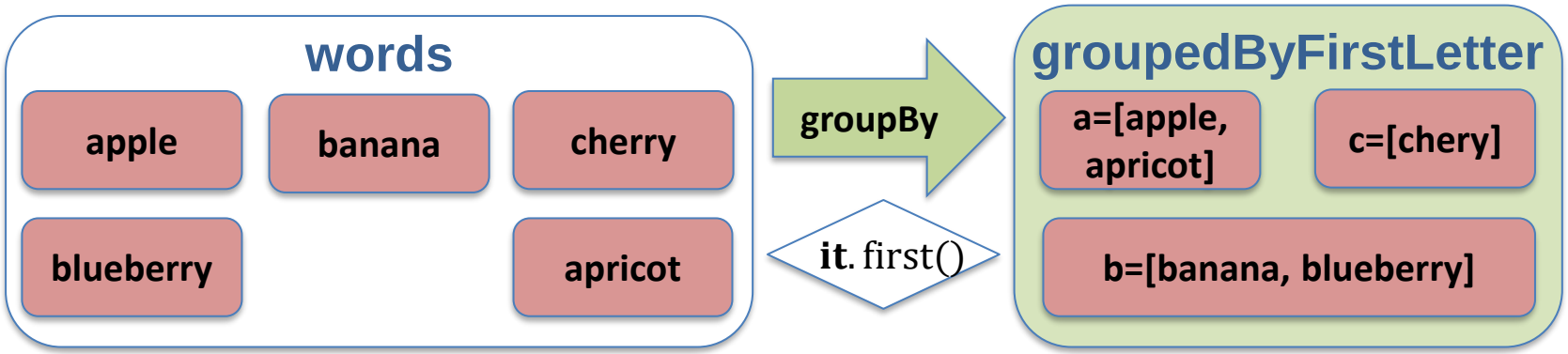
Podstawowa wersja **groupBy()** przyjmuje jako argument jedną funkcję lambda, która określa sposób tworzenia kluczy. W wyniku otrzymujemy mapę, gdzie:

- Klucz to wynik działania funkcji lambda na elementach kolekcji.
- Wartość to lista elementów przypisanych do danego klucza.

```
1  val words = listOf("apple", "banana", "cherry", "apricot", "blueberry")
2  val groupedByFirstLetter = words.groupBy { it.first() }
3
4  println(groupedByFirstLetter)
```

```
[11]
```

```
{a=[apple, apricot], b=[banana, blueberry], c=[cherry]}
```



Grupowanie

Wersja **groupBy()** z dwoma argumentami pozwala dodatkowo przekształcać wartości w mapie.

- Pierwsza funkcja lambda określa sposób tworzenia kluczy.
- Druga funkcja lambda określa, w jaki sposób przekształcać elementy przed umieszczeniem ich w wartościach mapy.

```
1  val words = listOf("apple", "banana", "cherry", "apricot", "blueberry")
2  val groupedWithLengths = words.groupBy(
3      keySelector = { it.first() },       // Klucz: pierwsza litera
4      valueTransform = { it.length }     // Wartość: długość słowa
5  )
6
7  println(groupedWithLengths)
8  // Wynik: {a=[5, 7], b=[6, 9], c=[6]}
   [13]
```

```
{a=[5, 7], b=[6, 9], c=[6]}
```

Uniwersytet Wrocławski

Łańcuchowanie metod

Łańcuchowanie metod polega na wywoływaniu kilku metod po sobie w sposób, który sprawia, że wynik jednej metody staje się wejściem dla kolejnej.

```
1  val numbers = listOf(5, 12, 3, 19, 8)
2
3  val result = numbers
4      .filter { it > 5 }            // Filtrujemy liczby większe niż 5
5      .map { it * 2 }               // Każdą liczbę mnożymy przez 2
6      .sorted()                     // Sortujemy rosnąco
7      .joinToString(" - ")          // Łączymy w ciąg znaków z separatorem "
       - "
8
9  println(result)
```

✓ [1] 1s 22ms

```
16 - 24 - 38
```