Uniwersytet Wrocławski

Rafał Lewandków

PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 1

WYKŁAD 1

- Podstawy Języka Kotlin
- Typy Danych
- Pętle, Wyrażenia, Instrukcje

Typy danych

- **Any** - pełni rolę *korzenia* hierarchii klas w tym języku. Jest to nadrzędna klasa wszystkich typów, które nie dopuszczają wartości **null**. Każda klasa w Kotlinie automatycznie dziedziczy po klasie **Any**. Umożliwia to uniwersalne podejście do operacji na obiektach, niezależnie od ich konkretnych typów.

```
1  fun printObject(obj: Any) {
2      println(obj.toString())
3  }
4
5  printObject(42)          // Wyjście: 42
6  printObject("Kotlin")    // Wyjście: Kotlin
   [1]
```

```
42
Kotlin
```

Typy danych

- **Nothing** - jest szczególnym typem w Kotlinie, który wskazuje na brak możliwości zwrócenia wartości.
  - Wszelkie wywołania funkcji zwracających **Nothing** nie prowadzą do kontynuacji kodu.
  - W hierarchii typów, jest podtypem każdego innego typu. Dzięki temu może być używany w sytuacjach, gdzie typ musi być określony, ale wiemy, że wartość nigdy nie będzie istniała.
  - Nie ma żadnych instancji.
  - Nie można stworzyć obiektu ani zmiennej typu **Nothing**.
  - **Nothing** to narzędzie, które pomaga wyrażać w kodzie koncepcję *braku wartości* w sytuacjach, gdzie program nie może kontynuować swojego działania w sposób standardowy.

Typy danych

- **Nothing** - jest szczególnym typem w Kotlinie, który wskazuje na brak możliwości zwrócenia wartości.

```kotlin
fun throwError(): Nothing {
    throw IllegalArgumentException("To jest błąd")
}
```

```kotlin
val result = when (value) {
    is Int -> value * 2
    is String -> value.length
    else -> throwError() // Rzucenie wyjątku jako `Nothing`
}
```

Typy danych

- **Unit** - jest szczególnym typem w Kotlinie, który wskazuje na brak możliwości zwrócenia wartości.
  - Głównym celem wprowadzenia typu **Unit** było zapewnienie spójnego podejścia do funkcji w paradygmacie programowania obiektowego i funkcyjnego. W Kotlinie wszystko jest bardziej zunifikowane, a nawet funkcje *nic niezwracające* są traktowane jako zwracające konkretny obiekt, czyli **Unit**.

  Cechy:
  - singleton - posiada tylko jedną instancję. Funkcje nieposiadające jawnie określonego typu zwracanego, zwracają instancję typu **Unit**.
  - niezmienność stanu - Każdorazowe wywołanie funkcji zwracającej **Unit** zawsze zwraca tą samą instancję
  - domyślny typ zwracany - Jeśli funkcja nie deklaruje jawnie typu zwracanego, domyślnie zwracany jest typ **Unit**

Typy danych

- **Unit** - jest szczególnym typem w Kotlinie, który wskazuje na brak możliwości zwrócenia wartości.

```
1  fun doSomething(): Unit {
2      println("Wykonuję operację!")
3  }
4
5  val result = doSomething() // result ma wartość Unit
[5]
```

```
Wykonuję operację!
```

```
1  fun doSomething() { // można pominąć typ zwracany
2      println("Wykonuję operację!")
3  }
4
5  val result = doSomething() // result ma wartość Unit
[6]
```

```
Wykonuję operację!
```

Typy danych

Niektóre różnice między **Nothing** i **Unit**

- **Unit** oznacza **brak użytecznej wartości zwracanej**..

- **Nothing** oznacza **brak możliwości zwrócenia wartości**..

- **Unit** ma jedną instancję. **Nothing** nie ma żadnych instancji.

Typy danych

Typy danych

Typ danych	Rozmiar (w bitach)	Zakres wartości	Uwagi
Byte	8	-128 do 127	Najmniejszy typ całkowitoliczbowy
Short	16	-32,768 do 32,767	Używany dla małych liczb całkowitych
Int	32	-2^{31} do $2^{31}-1$	Domyślny typ całkowity
Long	64	-2^{63} do $2^{63}-1$	Typ dla dużych liczb całkowitych
Float	32	~ ±3.4e-38 do ±3.4e38	Używany do liczb zmiennoprzecinkowych o pojedynczej precyzji
Double	64	~ ±1.7e-308 do ±1.7e308	Używany do liczb zmiennoprzecinkowych o podwójnej precyzji
Char	16	'\u0000' do '\uffff'	Jeden znak Unicode
Boolean	1 (teoretycznie)	`true` lub `false`	Wartości logiczne
String	Zmienna	Sekwencja znaków Unicode	Typ referencyjny
Array	Zmienna	Kolekcja elementów	Typ generyczny, np. `Array<Int>`
Any	-	Supertyp wszystkich typów w Kotlinie	Może przechowywać dowolny typ
Unit	-	Jedna wartość `Unit`	Odpowiada `void` w innych językach
Nothing	-	Nie zwraca wartości, brak instancji	Używany w funkcjach, które nigdy nie kończą działania normalnie

val vs **var**

W Kotlinie słowa kluczowe **val** (ang. *value*) i **var** (ang. *variable*) są używane do deklaracji zmiennych, różniąc się pod względem cech **mutowalności** (zmienności).

Słowo kluczowe **val**, oznacza **referencję** niemutowalną.

Oznacza to, że po przypisaniu wartości do zmiennej **val** nie można jej ponownie przypisać. Jednak **stan obiektu, na który wskazuje ta referencja, nadal może ulec zmianie**.

Uniwersytet Wrocławski

val vs **var**

val (ang. *value*) i **var** (ang. *variable*)

```
1  val a = 5
2  a = 2
   ✕ [23] 112ms
```

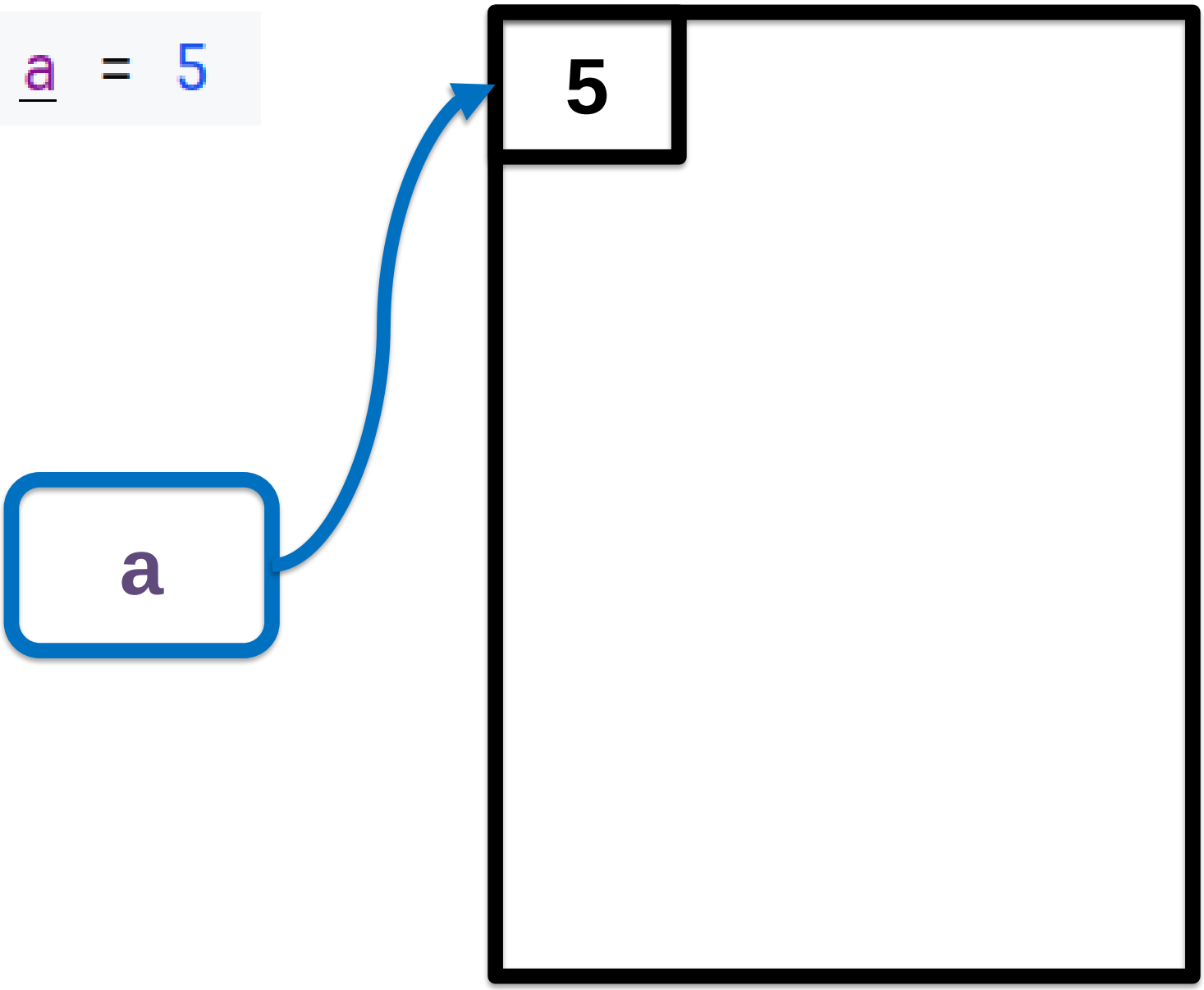
```
at Cell In[23], line 2, column 1: Val cannot be reassigned
```

```
1  var a = 5
2  a = 2
3  print(a)
   [9]
```

```
2
```

val vs var

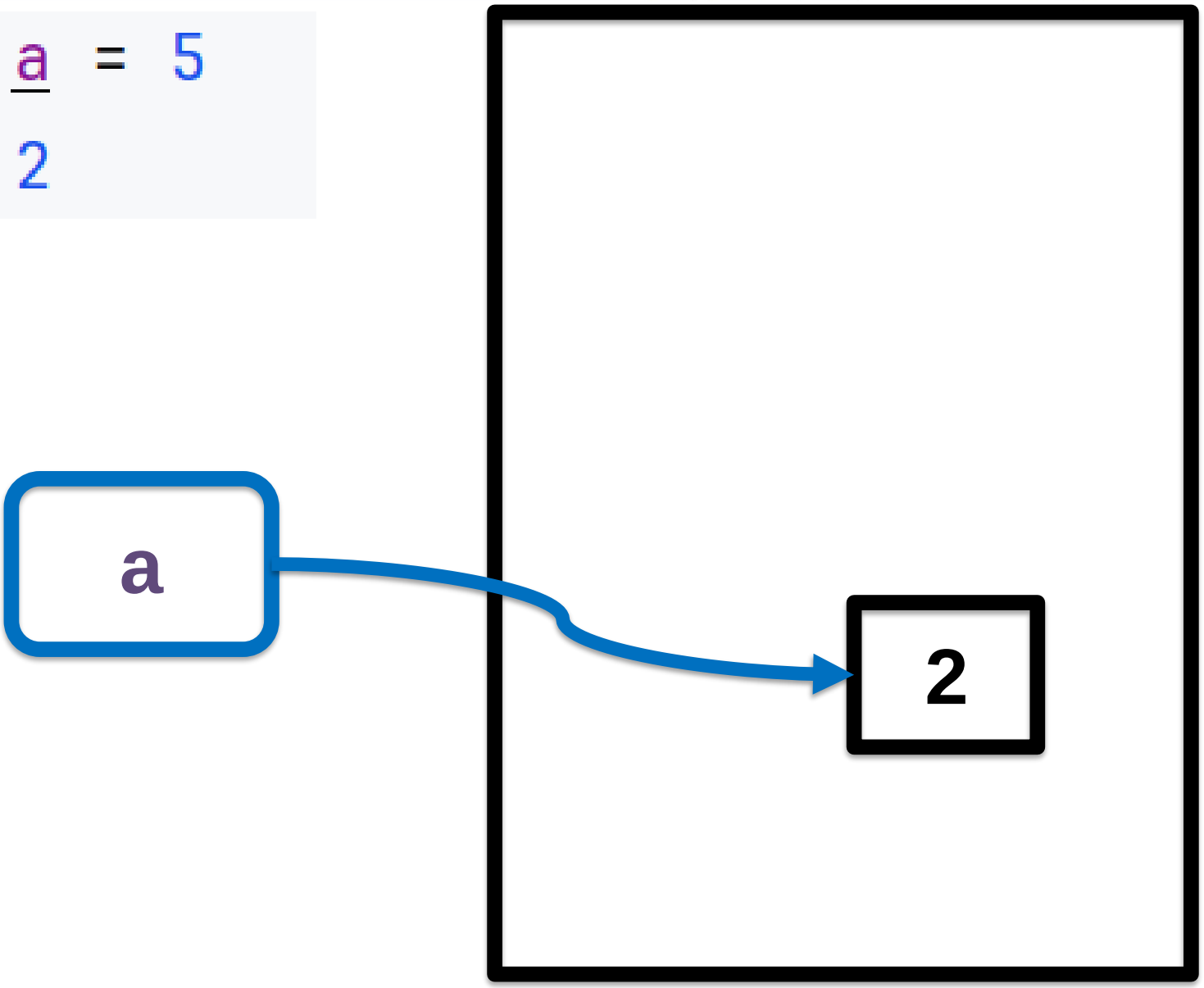
```
var a = 5
```



Uniwersytet Wrocławski

val vs var

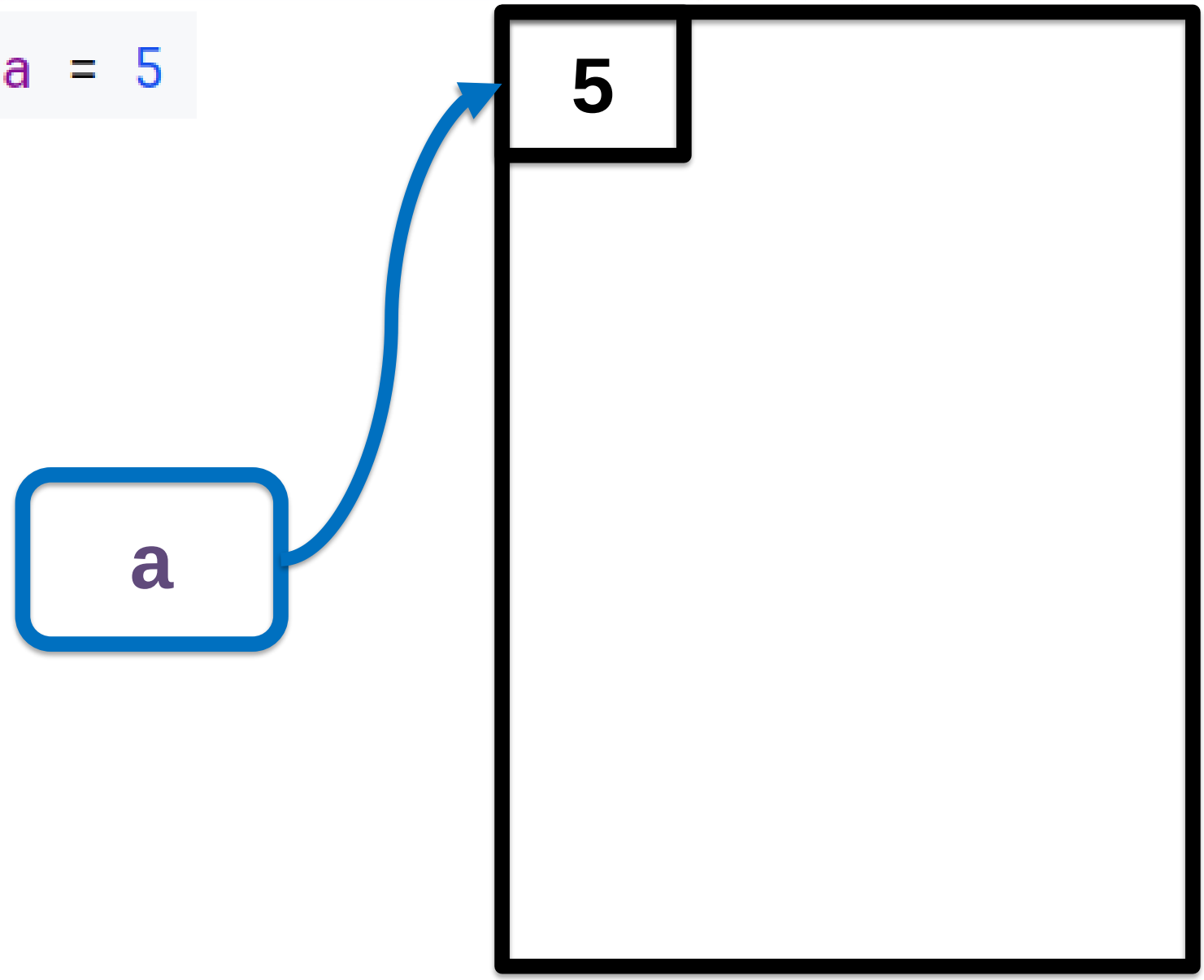
```
var a = 5

a = 2
```



Uniwersytet Wrocławski

val vs var

```
val a = 5
```

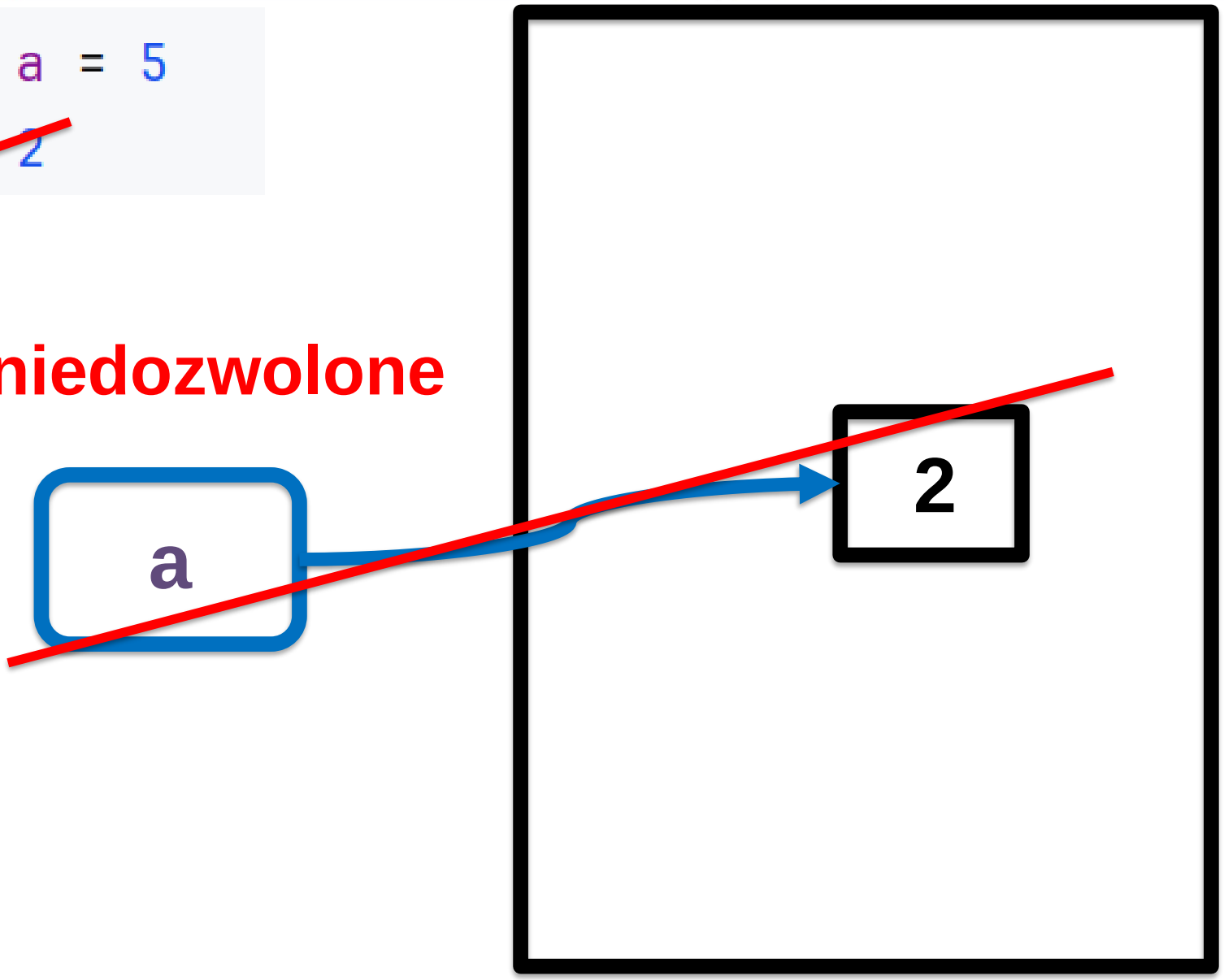


Uniwersytet Wrocławski

val vs var

```
val a = 5
a = 2
```

niedozwolone



Uniwersytet Wrocławski

val vs var

val (ang. *value*) i **var** (ang. *variable*)

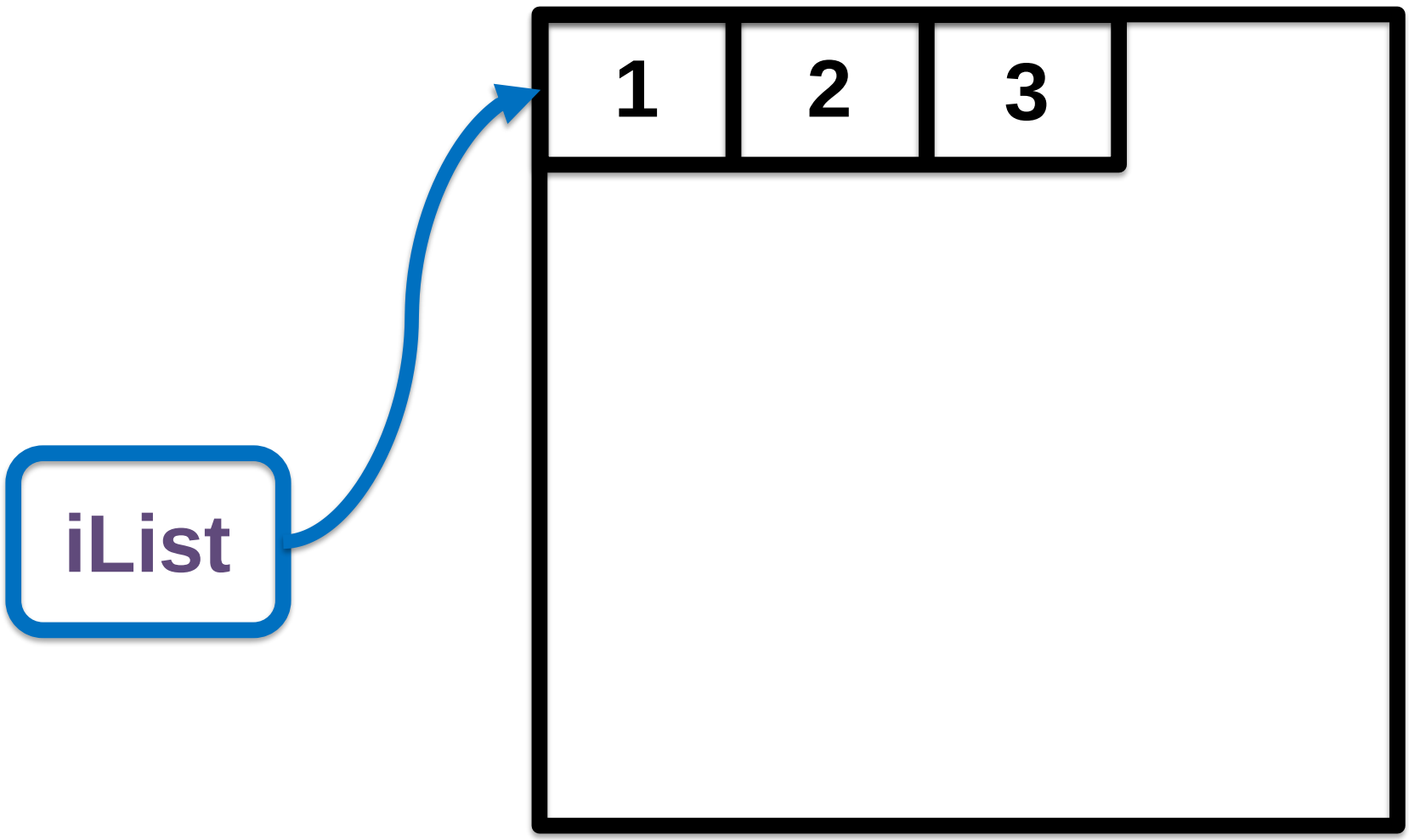
```
1  val immutableList = mutableListOf(1, 2, 3) // niemutowalna referencja
2  // 'wskazuje' mutowalny obiekt
3  immutableList.add(4) // Stan listy można zmienić
4  // immutableList = mutableListOf(5, 6) // Błąd: nie można
5  // przypisać nowej wartości do 'val'
6  print(immutableList)
   [8]
```

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

Uniwersytet Wrocławski

val vs var

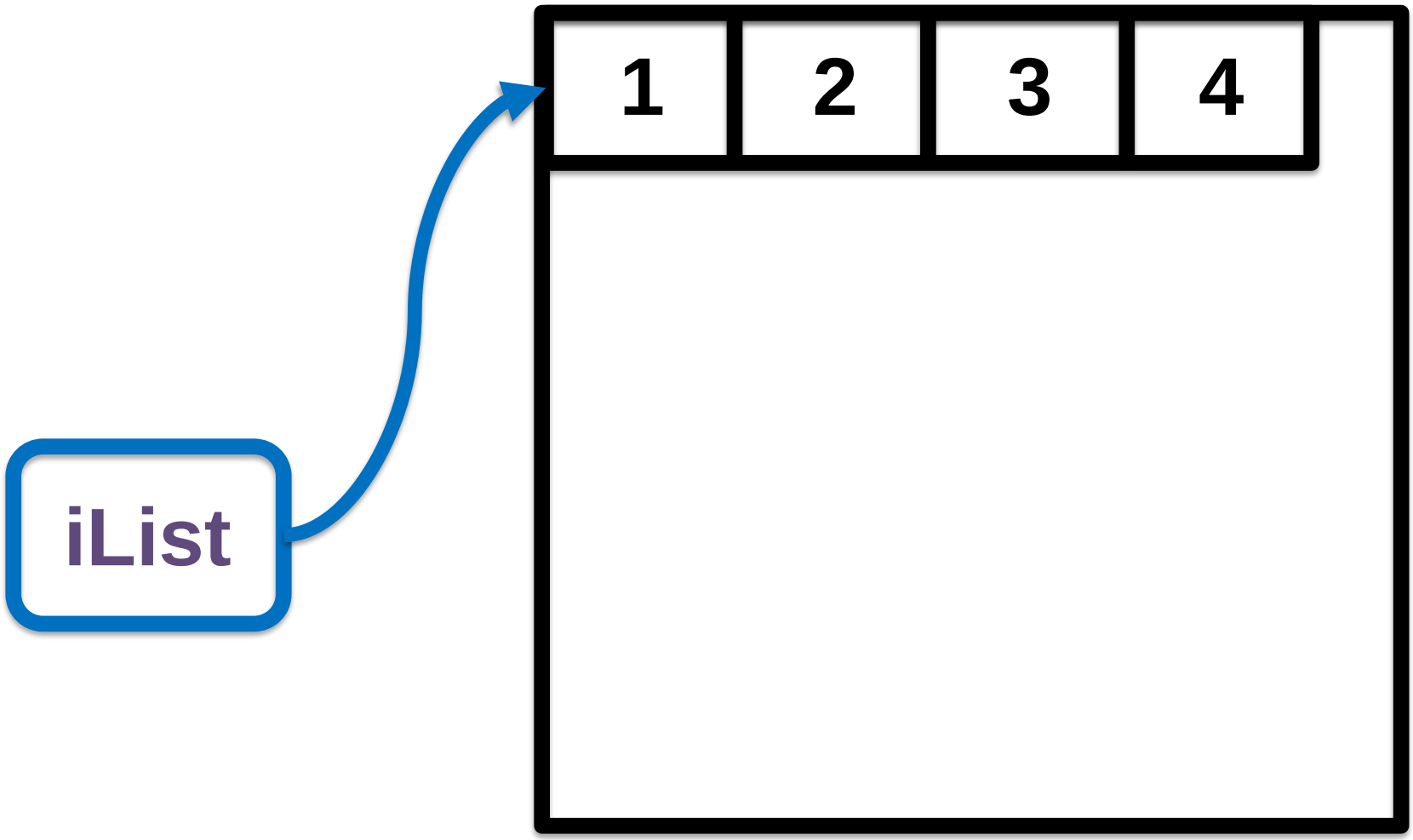
```
val iList = mutableListOf(1, 2, 3)
```



Uniwersytet Wrocławski

val vs var

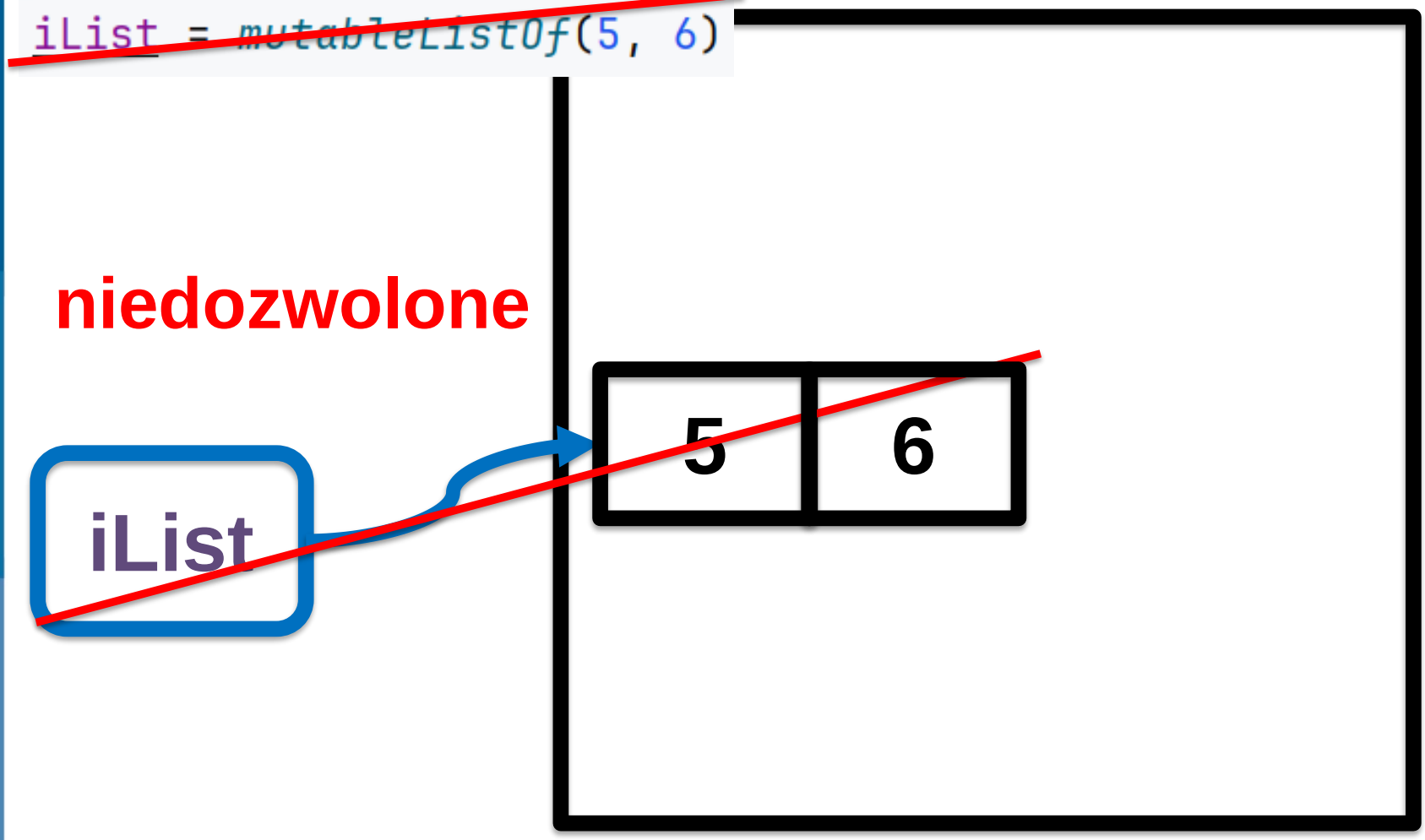
```
val iList = mutableListOf(1, 2, 3)
iList.add(4)
```



Uniwersytet Wrocławski

val vs var

```
val iList = mutableListOf(1, 2, 3)
iList.add(4)
iList = mutableListOf(5, 6)
```



Uniwersytet Wrocławski

val vs **const val**

const val i **val** mają różne zasady dotyczące przypisywania wartości.

Wartość stałej **const val** musi zostać przypisana w momencie kompilacji, co oznacza, że jej wartość musi być znana i ustalona już podczas kompilowania programu, a nie w czasie jego wykonywania.

Uniwersytet Wrocławski

val vs **const val**

const val i **val** mają różne zasady dotyczące przypisywania wartości.

Wartość stałej **const val** musi zostać przypisana w momencie kompilacji, co oznacza, że jej wartość musi być znana i ustalona już podczas kompilowania programu, a nie w czasie jego wykonywania.

Stała **const val** może przechowywać tylko wartości typu **String** lub typy *prymitywne* (np. **Int**, **Double**, **Boolean** itp.). Zatem nie może być przypisana do obiektu, funkcji, czy konstruktorów klas.

val vs **const val**

const val i **val** mają różne zasady dotyczące przypisywania wartości.

Wartość stałej **const val** musi zostać przypisana w momencie kompilacji, co oznacza, że jej wartość musi być znana i ustalona już podczas kompilowania programu, a nie w czasie jego wykonywania.

Stała **const val** może przechowywać tylko wartości typu **String** lub typy *prymitywne* (np. **Int**, **Double**, **Boolean** itp.). Zatem nie może być przypisana do obiektu, funkcji, czy konstruktorów klas.

Stała **const val** może być zadeklarowana na poziomie **globalnym**, czyli poza wszystkimi klasami, funkcjami czy innymi blokami kodu. Może być zadeklarowana również w obiekcie nazwanym (czyli klasyczny obiekt w Kotlinie) lub w obiekcie towarzyszącym (**companion object**)

Uniwersytet Wrocławski

val vs const val

const val jest używana, gdy wartość jest **stała**, **niezmienna** przez **cały czas życia programu** i znana na etapie **kompilacji**

Getter generuje nazwę pliku na podstawie **losowej wartości.** FILENAME **nie jest niemutowalny**, jest **read-only**.

Obiekt stowarzyszony

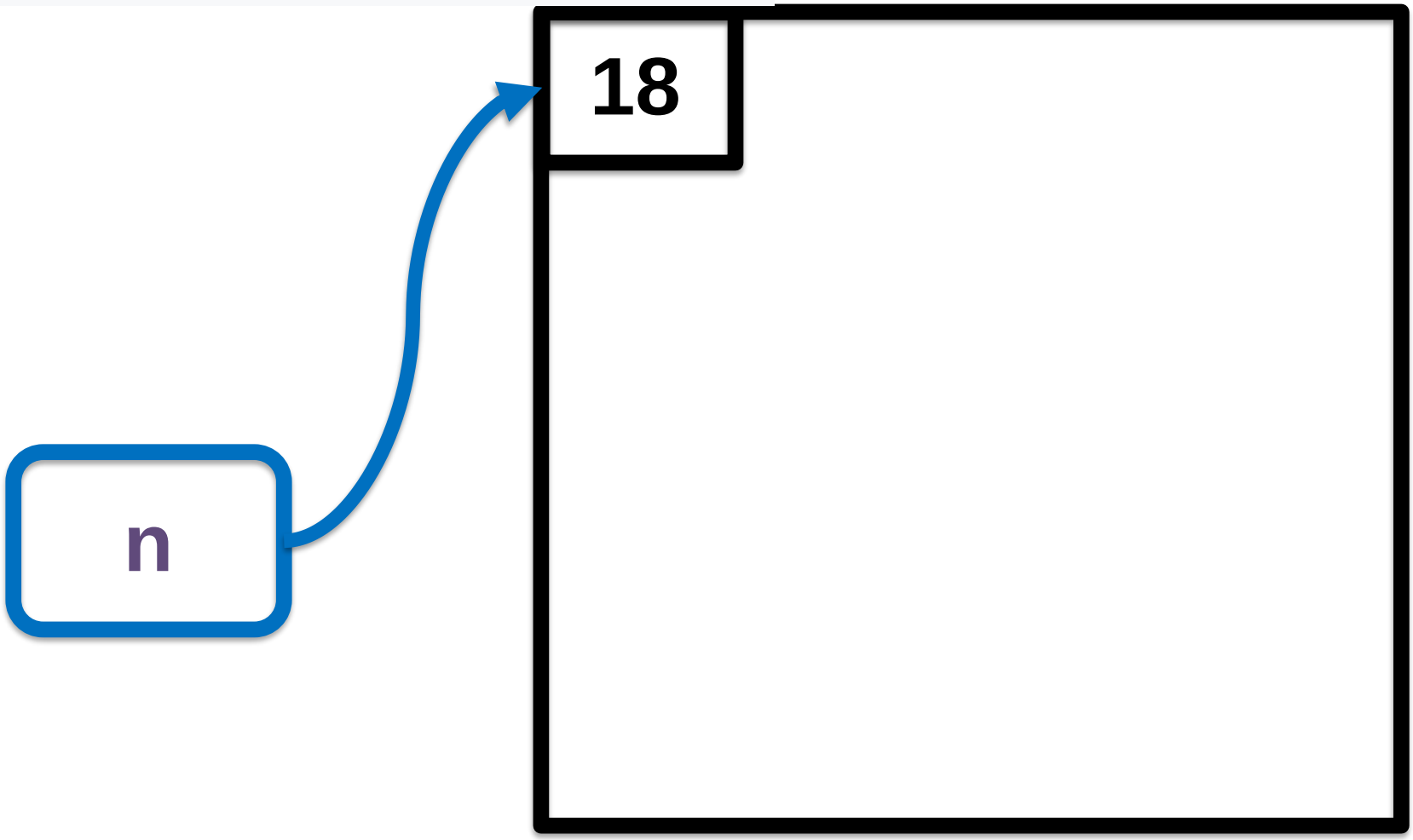
```
class YourClassName {
    companion object {
        const val FILE_EXTENSION = ".png"
        val FILENAME: String
            get() = "Img_" + (1 ≤ .. ≤ 100).random() + FILE_EXTENSION
    }
}

println(YourClassName.FILENAME)
println(YourClassName.FILENAME)
[11]
```

```
Img_42.png
Img_2.png
```

Uniwersytet Wrocławski

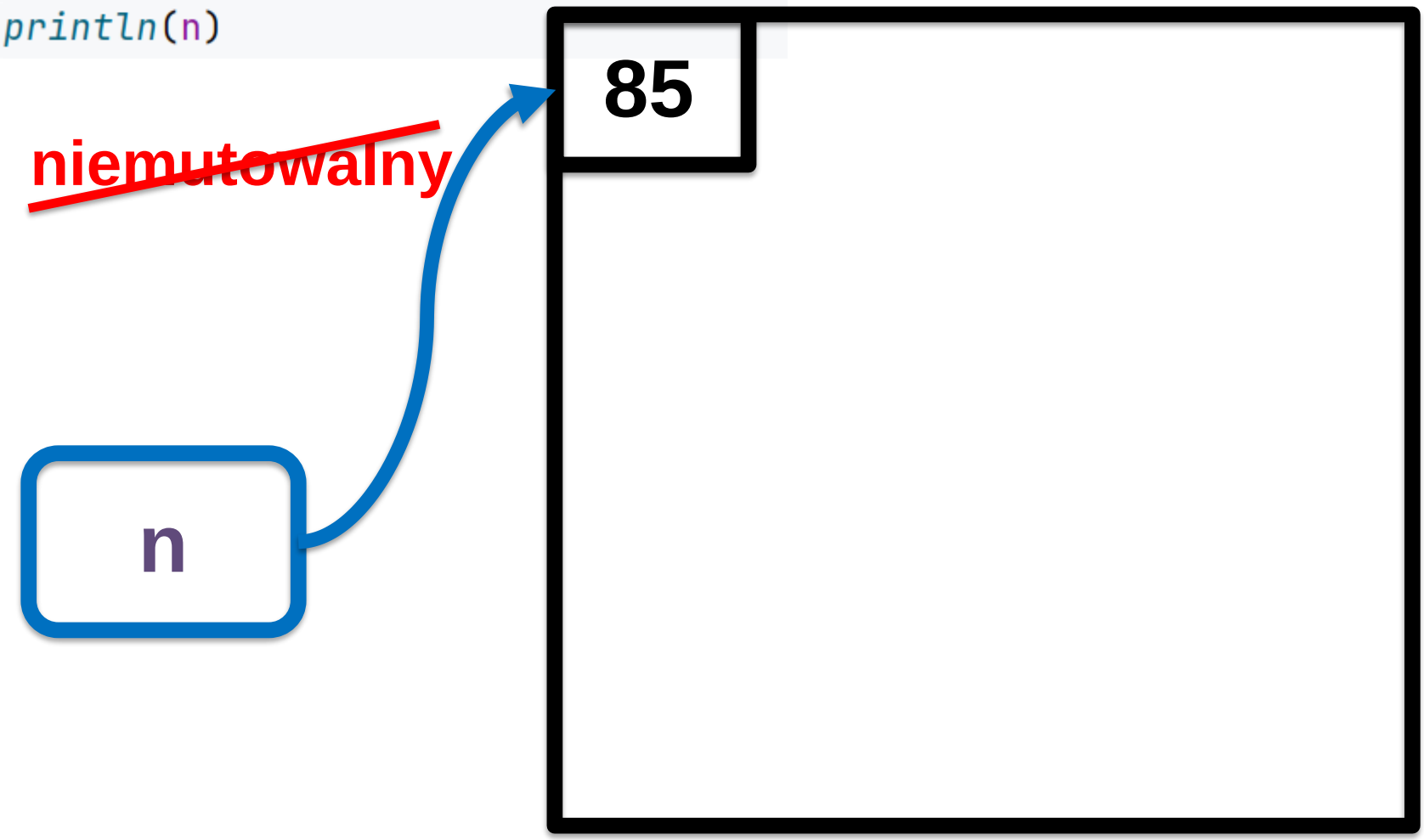
val vs var

```
val n: Int
    get() = (0 ≤ .. ≤ 100).random()
println(n)
```



val vs var

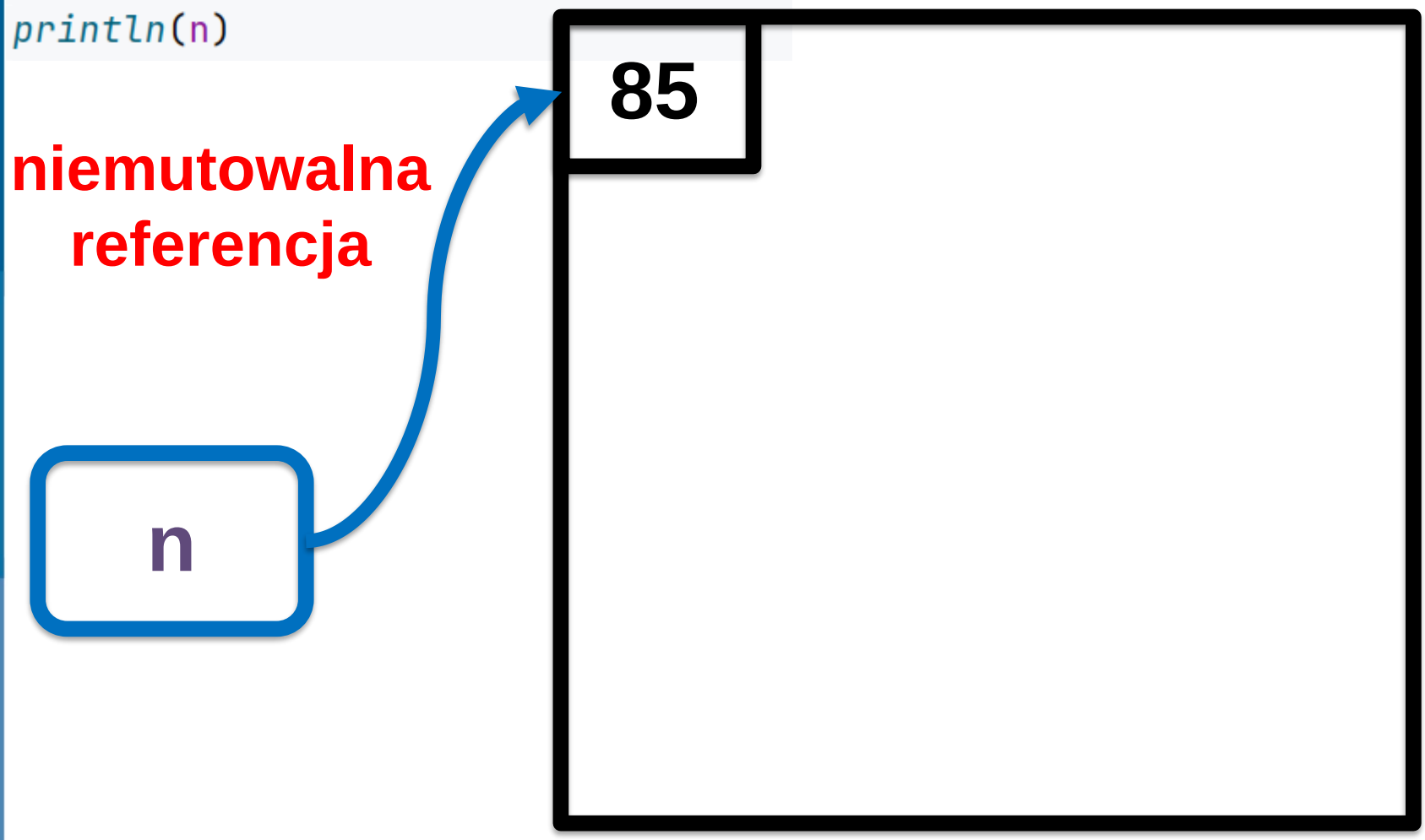
```
val n: Int
    get() = (0 ≤ .. ≤ 100).random()
println(n)
println(n)
```



Uniwersytet Wrocławski

val vs var

```
val n: Int
    get() = (0 ≤ .. ≤ 100).random()
println(n)
println(n)
```

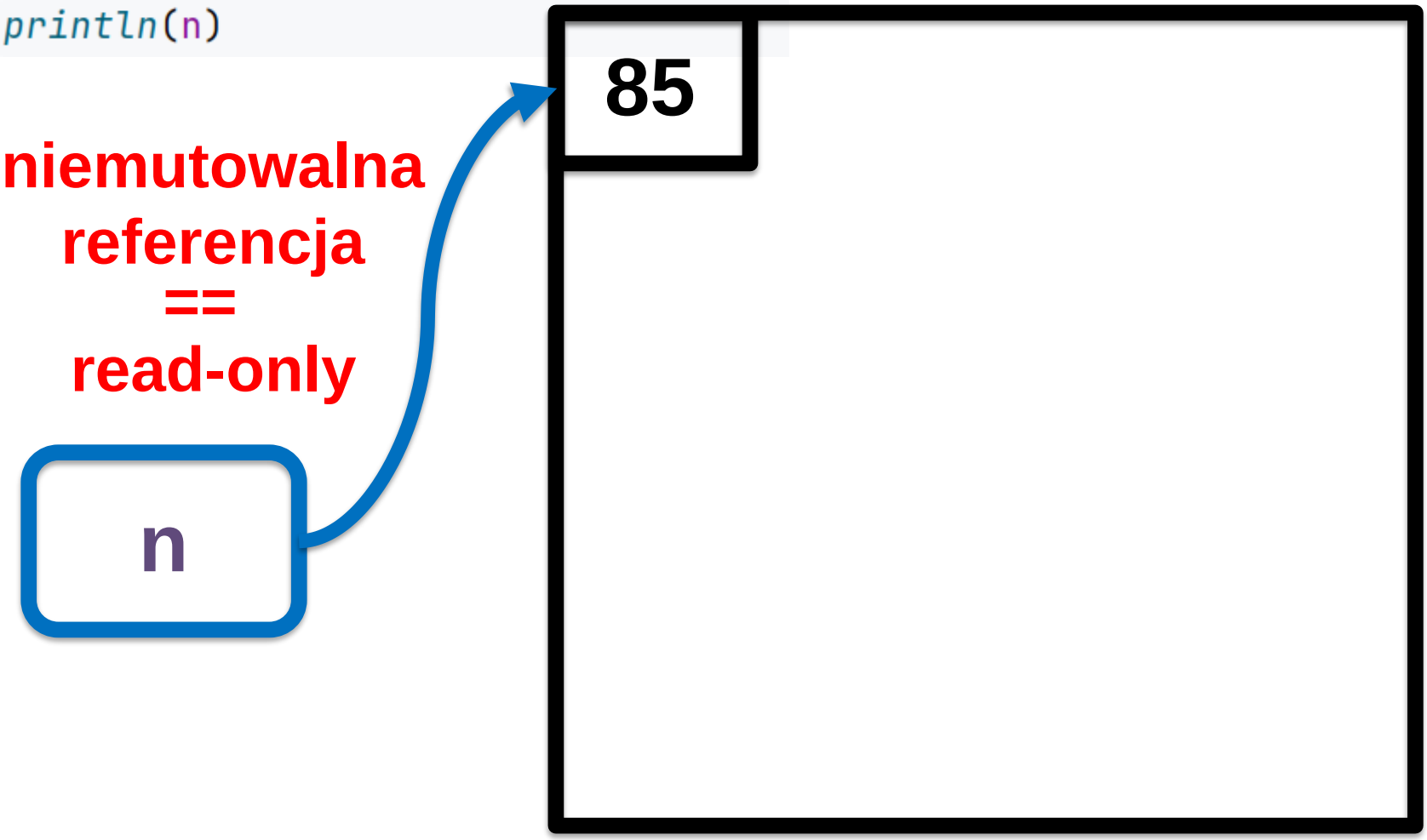


Uniwersytet Wrocławski

val vs var

```
val n: Int
    get() = (0 ≤ .. ≤ 100).random()
println(n)
println(n)
```

niemutowalna referencja == read-only



Uniwersytet Wrocławski

Typy **nullable**

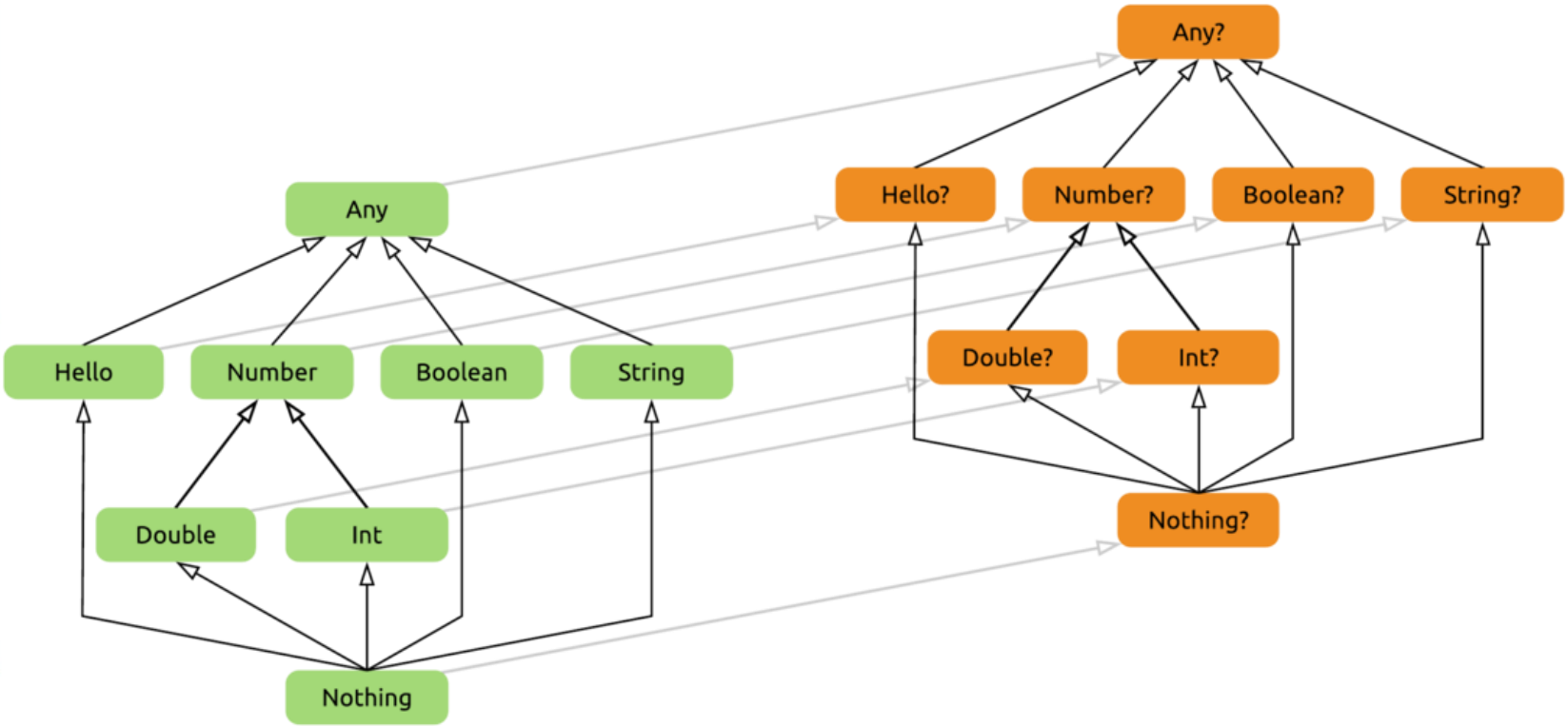
W Kotlinie bezpieczeństwo związane z **null** (ang. *null safety*) to kluczowa cecha, która została wprowadzona w celu minimalizacji błędów związanych z odwołaniami do **null**. Funkcjonalność ta zmniejsza ryzyko wystąpienia wyjątków **NullPointerException**.

Typ w Kotlinie może być oznaczony jako **nullable**, jeśli może przyjmować wartość **null**. Aby to zrobić, dodajemy znak **?** po nazwie typu.

```
1  val name: String? = null  // Zmienna może być null
2  val nonNullableName: String = "Kotlin"  // Zmienna nie może być null
```

Uniwersytet Wrocławski

Typy **nullable**



https://zsmb.co/images/typical-kotlin/typical_nothing.png

Uniwersytet Wrocławski

Typy **nullable**

Wartość ***nullable*** zainicjowana **null**

```
1  val name: String? = null
2  val length: Int? = name?.length
3  println(length)
4
5  val anotherName: String? = "Kotlin"
6  println(anotherName?.length)
```

Typy **nullable**

Operator **bezpiecznego wywołania,** sprawdza, czy obiekt jest **null**. Jeśli jest, operacja zostanie **pominięta**, a wynik zwróci **"null".**

Wartość ***nullable*** zainicjowana **null**

```
1  val name: String? = null
2  val length: Int? = name?.length
3  println(length)
4
5  val anotherName: String? = "Kotlin"
6  println(anotherName?.length)
```

Uniwersytet Wrocławski

Typy **nullable**

Wartość ***nullable*** zainicjowana **null**

Operator **bezpiecznego wywołania,** sprawdza, czy obiekt jest **null**. Jeśli jest, operacja zostanie **pominięta**, a wynik zwróci **"null".**

```
1  val name: String? = null
2  val length: Int? = name?.length
3  println(length)
4
5  val anotherName: String? = "Kotlin"
6  println(anotherName?.length)
```

Próba dostępu do właściwości obiektu **zainicjowanego null**, zwróci **„null”**

Uniwersytet Wrocławski

Typy **nullable**

Operator **bezpiecznego wywołania,** sprawdza, czy obiekt jest **null**. Jeśli jest, operacja zostanie **pominięta**, a wynik zwróci **"null".**

Wartość ***nullable*** zainicjowana **null**

```
1 val name: String? = null
2 val length: Int? = name?.length
3 println(length)
4
5 val anotherName: String? = "Kotlin"
6 println(anotherName?.length)
```

Próba dostępu do właściwości obiektu **zainicjowanego null**, zwróci **„null”**

Wartość ***nullable*** zainicjowana **null**

```
1 val name: String? = null
2 val length: Int = name!!.length  // Rzuci NullPointerException
[2]
> Stacktrace…
java.lang.NullPointerException: null
at Cell In[2], line 2
```

Typy **nullable**

Operator **bezpiecznego wywołania,** sprawdza, czy obiekt jest **null**. Jeśli jest, operacja zostanie **pominięta**, a wynik zwróci **"null".**

Wartość ***nullable*** zainicjowana **null**

```
1  val name: String? = null
2  val length: Int? = name?.length
3  println(length)
4
5  val anotherName: String? = "Kotlin"
6  println(anotherName?.length)
```

Próba dostępu do właściwości obiektu **zainicjowanego null**, zwróci **„null”**

Operator **!! wymusza** na kompilatorze **założenie**, że obiekt na pewno **nie jest null.**

Wartość ***nullable*** zainicjowana **null**

```
1  val name: String? = null
2  val length: Int = name!!.length  // Rzuci NullPointerException
[2]
```

```
> Stacktrace…
java.lang.NullPointerException: null
at Cell In[2], line 2
```

Uniwersytet Wrocławski

Typy **nullable**

Operator **bezpiecznego wywołania,** sprawdza, czy obiekt jest **null**. Jeśli jest, operacja zostanie **pominięta**, a wynik zwróci **"null".**

Wartość ***nullable*** zainicjowana **null**

```
1  val name: String? = null
2  val length: Int? = name?.length
3  println(length)
4
5  val anotherName: String? = "Kotlin"
6  println(anotherName?.length)
```

Próba dostępu do właściwości obiektu **zainicjowanego null**, zwróci **„null”**

Operator **!! wymusza** na kompilatorze **założenie**, że obiekt na pewno **nie jest null.**

Wartość ***nullable*** zainicjowana **null**

```
1  val name: String? = null
2  val length: Int = name!!.length  // Rzuci NullPointerException
[2]
> Stacktrace…
java.lang.NullPointerException: null
at Cell In[2], line 2
```

Próba dostępu do właściwości obiektu **zainicjowanego null**, zakończy się rzuceniem wyjątku

Uniwersytet Wrocławski

Typy **nullable**

Operator	Zalety
`?.` (bezpieczny operator wywołania)	• Eliminuje ryzyko `NullPointerException` • Czytelny sposób na obsługę `null` • Umożliwia zagnieżdżone wywołania
`!!` (wymuszenie nie-`null`)	• Przydatny, gdy jesteśmy pewni, że wartość nigdy nie będzie `null` • Prosty i szybki w użyciu

Operator	Wady
`?.` (bezpieczny operator wywołania)	• Może prowadzić do sytuacji, gdzie `null` propaguje się nieoczekiwanie.
`!!` (wymuszenie nie-`null`)	• Ryzykowny w użyciu, jeśli wartość może być `null`. • Może prowadzić do wyjątków w czasie działania programu.

Typy **nullable**

Podsumowanie:

- **Null safety** w Kotlinie eliminuje błędy związane z **null** poprzez wymuszenie obsługi zmiennych *nullable*.

Typy **nullable**

Podsumowanie:

- **Null safety** w Kotlinie eliminuje błędy związane z **null** poprzez wymuszenie obsługi zmiennych *nullable*.

- Bezpieczne wywołanie (**?.**) umożliwia bezpieczne operacje na zmiennych *nullable*.

Typy **nullable**

Podsumowanie:

- **Null safety** w Kotlinie eliminuje błędy związane z **null** poprzez wymuszenie obsługi zmiennych *nullable*.

- Bezpieczne wywołanie (**?.**) umożliwia bezpieczne operacje na zmiennych *nullable*.

- Operator Elvis (**?:**) pozwala ustawić wartość domyślną w przypadku **null**.

```
val length: Int = name?.length ?: -1
```

Typy **nullable**

Podsumowanie:

- **Null safety** w Kotlinie eliminuje błędy związane z **null** poprzez wymuszenie obsługi zmiennych *nullable*.

- Bezpieczne wywołanie (**?.**) umożliwia bezpieczne operacje na zmiennych *nullable*.

- Operator Elvis (**?:**) pozwala ustawić wartość domyślną w przypadku **null**.

```
val length: Int = name?.length ?: -1
```

- Rzutowanie bezpieczne (**as?**) zwraca **null** w przypadku nieudanego rzutowania.

```
val cat: String? = animal as? String
```

Typy **nullable**

Podsumowanie:

- **Null safety** w Kotlinie eliminuje błędy związane z **null** poprzez wymuszenie obsługi zmiennych *nullable*.

- Bezpieczne wywołanie (**?.**) umożliwia bezpieczne operacje na zmiennych *nullable*.

- Operator Elvis (**?:**) pozwala ustawić wartość domyślną w przypadku **null**.

```
val length: Int = name?.length ?: -1
```

- Rzutowanie bezpieczne (**as?**) zwraca **null** w przypadku nieudanego rzutowania.

```
val cat: String? = animal as? String
```

- Wymuszenie *nie-null* (**!!**) powinno być używane ostrożnie, ponieważ może prowadzić do wyjątków.

Pętle i instrukcje

W Kotlinie wyrażenie **if** działa podobnie jak w wielu innych językach programowania, ale ma jedną istotną różnicę: jest **wyrażeniem**, co oznacza, że może zwracać wartość. Dzięki temu może być używane zarówno jako standardowa instrukcja warunkowa (z ciałem blokowym), jak i jako element przypisania lub wyrażenie (ciało wyrażeniowe).

Pętle i instrukcje


```
1  val liczba = 10
2
3  if (liczba > 5) {
4      println("Liczba jest większa niż 5")
5  } else {
6      println("Liczba jest mniejsza lub równa 5")
7  }
```

✓ [1] 305ms

```
Liczba jest większa niż 5
```

```
1  val liczba = 10
2
3  val wynik = if (liczba > 5) "Większa" else "Mniejsza lub równa"
4
5  println("Liczba jest: $wynik")  // Wydrukuje: Liczba jest: Większa
```

✓ [3] 104ms

```
Liczba jest: Większa
```

Pętle i instrukcje

Warunek

Blok kodu wywoływany przy **spełnieniu** warunku

```
1  val liczba = 10
2
3  if (liczba > 5) {
4      println("Liczba jest większa niż 5")
5  } else {
6      println("Liczba jest mniejsza lub równa 5")
7  }
```

✓ [1] 305ms

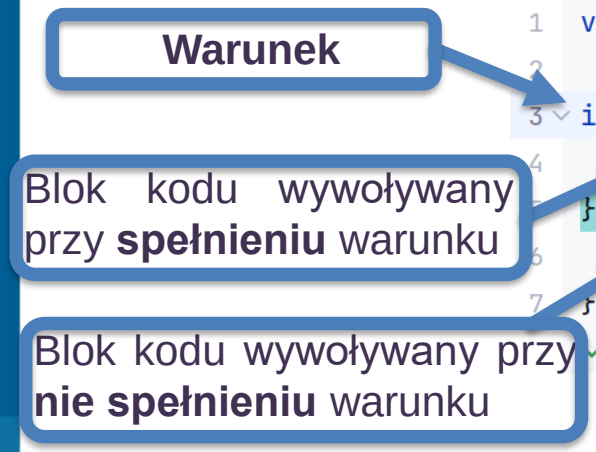
```
Liczba jest większa niż 5
```

```
1  val liczba = 10
2
3  val wynik = if (liczba > 5) "Większa" else "Mniejsza lub równa"
4
5  println("Liczba jest: $wynik")  // Wydrukuje: Liczba jest: Większa
```

✓ [3] 104ms

```
Liczba jest: Większa
```

Pętle i instrukcje




```
1  val liczba = 10
2
3  if (liczba > 5) {
4      println("Liczba jest większa niż 5")
5  } else {
6      println("Liczba jest mniejsza lub równa 5")
7  }
```

✓ [1] 305ms

```
Liczba jest większa niż 5
```

```
1  val liczba = 10
2
3  val wynik = if (liczba > 5) "Większa" else "Mniejsza lub równa"
4
5  println("Liczba jest: $wynik")  // Wydrukuje: Liczba jest: Większa
```

✓ [3] 104ms

```
Liczba jest: Większa
```

Pętle i instrukcje

Warunek

Blok kodu wywoływany przy **spełnieniu** warunku

Blok kodu wywoływany przy **nie spełnieniu** warunku

```
1  val liczba = 10
2
3  if (liczba > 5) {
4      println("Liczba jest większa niż 5")
5  } else {
6      println("Liczba jest mniejsza lub równa 5")
7  }
```

✓ [1] 305ms

```
Liczba jest większa niż 5
```

Warunek **if** z ciałem **wyrażeniowym**

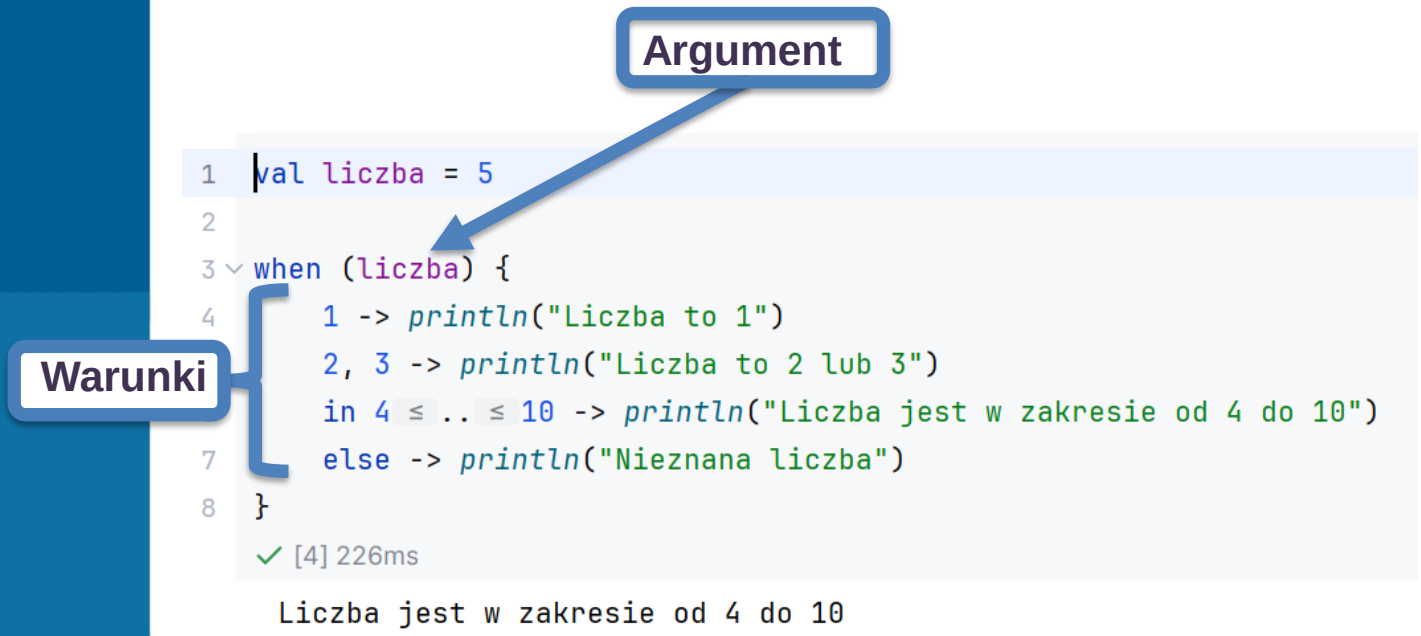
```
1  val liczba = 10
2
3  val wynik = if (liczba > 5) "Większa" else "Mniejsza lub równa"
4
5  println("Liczba jest: $wynik")  // Wydrukuje: Liczba jest: Większa
```

✓ [3] 104ms

```
Liczba jest: Większa
```

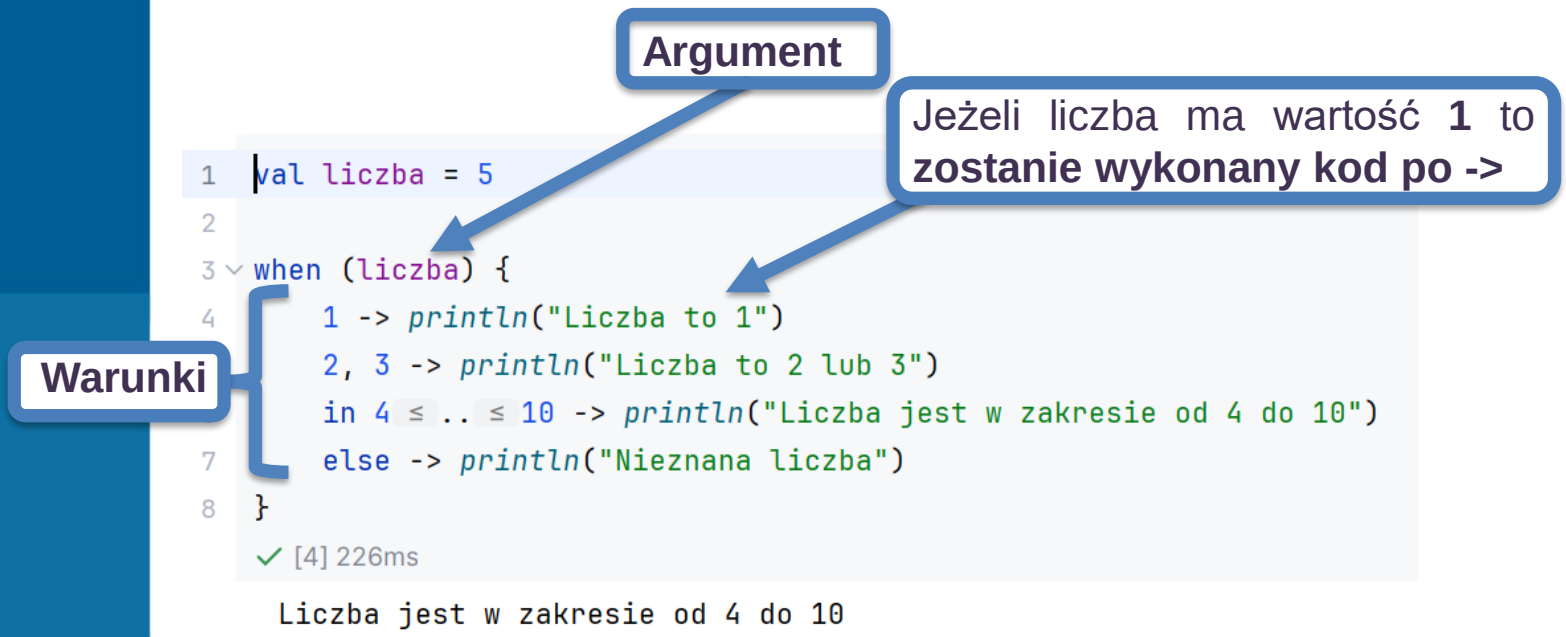
Pętle i instrukcje

Wyrażenie **when** w Kotlinie to konstruktor warunkowy, który zastępuje instrukcję **switch** znaną z innych języków.



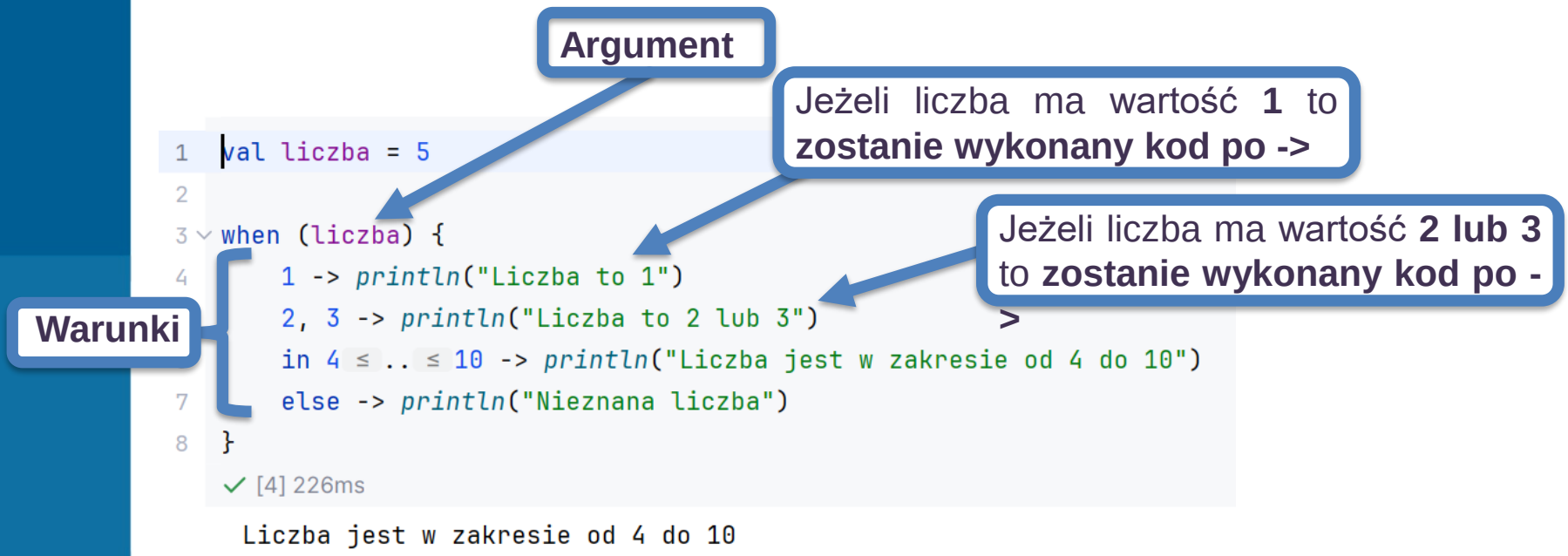
Pętle i instrukcje

Wyrażenie **when** w Kotlinie to konstruktor warunkowy, który zastępuje instrukcję **switch** znaną z innych języków.



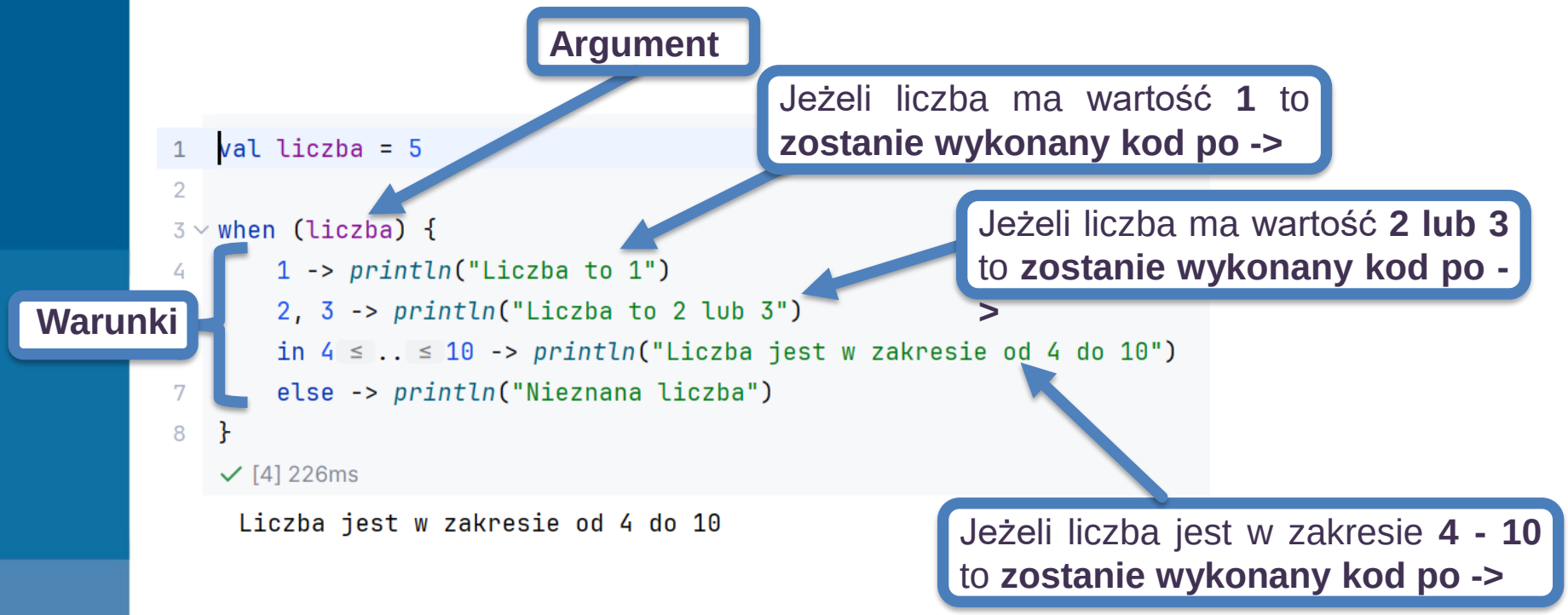
Pętle i instrukcje

Wyrażenie **when** w Kotlinie to konstruktor warunkowy, który zastępuje instrukcję **switch** znaną z innych języków.



Pętle i instrukcje

Wyrażenie **when** w Kotlinie to konstruktor warunkowy, który zastępuje instrukcję **switch** znaną z innych języków.



Pętle i instrukcje

Wyrażenie **when** w Kotlinie to konstruktor warunkowy, który zastępuje instrukcję **switch** znaną z innych języków.

Pętle i instrukcje

Wyrażenie **when** w Kotlinie to konstruktor warunkowy, który zastępuje instrukcję **switch** znaną z innych języków.

Pętle i instrukcje

Wyrażenie **when** w Kotlinie to konstruktor warunkowy, który zastępuje instrukcję **switch** znaną z innych języków.

Wyrażenie bezargumentowe

```
1  val x = 15
2
   when {
       x % 2 == 0 -> println("Liczba parzysta")
       x % 2 != 0 -> println("Liczba nieparzysta")
       x > 10 -> println("Większa niż 10")
   }
```

✓ [6] 144ms

```
Liczba nieparzysta
```

Każdy warunek to **osobne wyrażenie logiczne**. Blok **when** sprawdza warunki w kolejności, a kod jest wykonywany **przy pierwszym spełnionym warunku**.

Pętle i instrukcje

Najczęściej używana jest do iteracji przez zakresy (**ranges**) lub kolekcje (**collections**). Podstawowa składnia

```
for (element in kolekcja) {
    // Kod wykonywany dla każdego elementu
}
```

Pętle i instrukcje

Najczęściej używana jest do iteracji przez zakresy (**ranges**) lub kolekcje (**collections**). Podstawowa składnia

Pętla for wykonana od 1 do 5 **włącznie**.

```
1 for (i in 1 ≤ .. ≤ 5) {
2     println("Liczba: $i")
3 }
```

✓ [12] 80ms

```
Liczba: 1
Liczba: 2
Liczba: 3
Liczba: 4
Liczba: 5
```

Pętle i instrukcje

Najczęściej używana jest do iteracji przez zakresy (**ranges**) lub kolekcje (**collections**). Podstawowa składnia

Pętla for wykonana od 5 do 1 **włącznie**.

```
1 for (i in 5 ≥ downTo ≥ 1) {
2     println("Liczba: $i")
3 }
```

✓ [13] 86ms

```
Liczba: 5
Liczba: 4
Liczba: 3
Liczba: 2
Liczba: 1
```

Pętle i instrukcje

Najczęściej używana jest do iteracji przez zakresy (**ranges**) lub kolekcje (**collections**). Podstawowa składnia

Pętla for wykonana od 0 do 10 **włącznie,** z krokiem 2.

```
1 for (i in 0 ≤ .. ≤ 10 step 2) {
2     println("Liczba: $i")
3 }
```

✓ [15] 81ms

```
Liczba: 0
Liczba: 2
Liczba: 4
Liczba: 6
Liczba: 8
Liczba: 10
```

Pętle i instrukcje

Najczęściej używana jest do iteracji przez zakresy (**ranges**) lub kolekcje (**collections**). Podstawowa składnia

Pętla for wykonana od 1 do 5.

```
1 for (i in 1 ≤ until < 5) {
2     println("Liczba: $i")
3 }
```

✓ [16] 88ms

```
Liczba: 1
Liczba: 2
Liczba: 3
Liczba: 4
```

Pętle i instrukcje

Najczęściej używana jest do iteracji przez zakresy (**ranges**) lub kolekcje (**collections**). Podstawowa składnia

- **1..5** – liczby od 1 do 5.
- **5 downTo 1** – liczby od 5 do 1.
- **1..10 step 2** – liczby od 1 do 10, co drugą.
- **1 until 5** – liczby od 1 do 4 (bez 5).

Pętle i instrukcje

Najczęściej używana jest do iteracji przez zakresy (**ranges**) lub kolekcje (**collections**). Podstawowa składnia

```
1  val lista = listOf("Jabłko", "Banan", "Czereśnia")
2  for (owoc in lista) {
3      println("Owoc: $owoc")
4  }
```

✓ [17] 137ms

```
Owoc: Jabłko
Owoc: Banan
Owoc: Czereśnia
```

Pętle i instrukcje

Najczęściej używana jest do iteracji przez zakresy (**ranges**) lub kolekcje (**collections**). Podstawowa składnia

```
1  val lista = listOf("Jabłko", "Banan", "Czereśnia")
2  for (indeks in lista.indices) {
3      println("Indeks $indeks, wartość: ${lista[indeks]}")
4  }
```

✓ [18] 488ms

```
Indeks 0, wartość: Jabłko
Indeks 1, wartość: Banan
Indeks 2, wartość: Czereśnia
```

Pętle i instrukcje

Najczęściej używana jest do iteracji przez zakresy (**ranges**) lub kolekcje (**collections**). Podstawowa składnia

```
1  val mapa = mapOf(1 to "Jeden", 2 to "Dwa", 3 to "Trzy")
2  for ((klucz, wartość) in mapa) {
3      println("Klucz: $klucz, Wartość: $wartość")
4  }
```

✓ [19] 373ms

```
Klucz: 1, Wartość: Jeden
Klucz: 2, Wartość: Dwa
Klucz: 3, Wartość: Trzy
```

Pętle i instrukcje

```
1  var licznik = 5
2
3  while (licznik > 0) {
4      println("Licznik: $licznik")
5      licznik--
6  }
```

✓ [20] 184ms

```
Licznik: 5
Licznik: 4
Licznik: 3
Licznik: 2
Licznik: 1
```

Pętle i instrukcje

```
1  var licznik = 0
2
3  do {
4      println("Licznik: $licznik")
5      licznik++
6  } while (licznik < 3)
```

✓ [21] 94ms

```
Licznik: 0
Licznik: 1
Licznik: 2
```

Pętle i instrukcje

```
1  var licznik = 10
2
3  while (licznik > 0) {
4      if (licznik == 5) {
5          println("Przerywam pętlę.")
6          break  // Kończy działanie pętli
7      }
8      if (licznik % 2 == 0) {
9          licznik--
10         continue  // Przechodzi do następnej iteracji
11     }
12     println("Licznik: $licznik")
13     licznik--
14 }
```

✓ [22] 107ms

```
Licznik: 9
Licznik: 7
Przerywam pętlę.
```