



# PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 1

## WYKŁAD 7

- Inicjalizacja i Delegacja

# Inicjalizacja opóźniona

Gdy nie możemy lub nie chcemy od razu przypisać wartości do zmiennej w momencie jej deklaracji, możemy wykorzystać **opóźnioną inicjalizację**.

**lateinit** to modyfikator, który pozwala na opóźnioną inicjalizację właściwości zmiennych **tylko dla typu var**.

Modyfikator pozwalający  
na **późną inicjalizację**



```
3 lateinit var lateinitUser : User
```

# Inicjalizacja opóźniona

Gdy nie możemy lub nie chcemy od razu przypisać wartości do zmiennej w momencie jej deklaracji, możemy wykorzystać **opóźnioną inicjalizację**.

**lateinit** to modyfikator, który pozwala na opóźnioną inicjalizację właściwości zmiennych **tylko dla typu var**.

Modyfikator pozwalający  
na **późną inicjalizację**

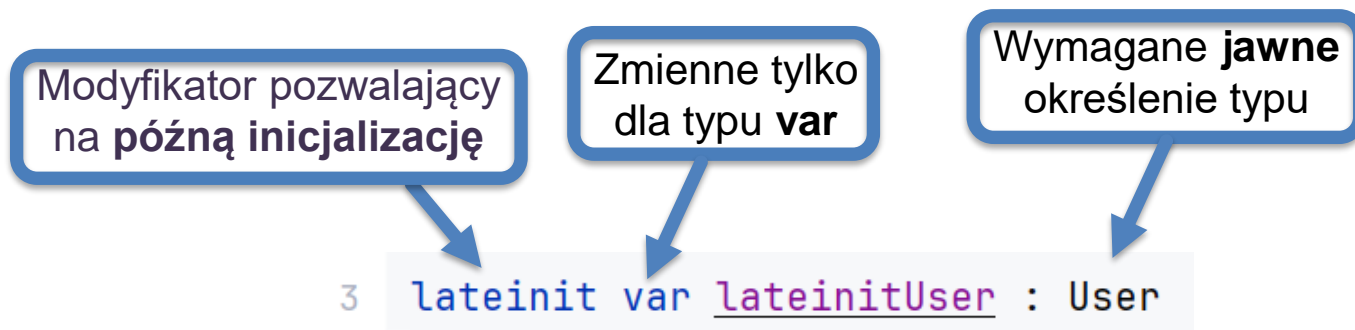
Zmienne tylko  
dla typu **var**

```
3 lateinit var lateinitUser : User
```

# Inicjalizacja opóźniona

Gdy nie możemy lub nie chcemy od razu przypisać wartości do zmiennej w momencie jej deklaracji, możemy wykorzystać **opóźnioną inicjalizację**.

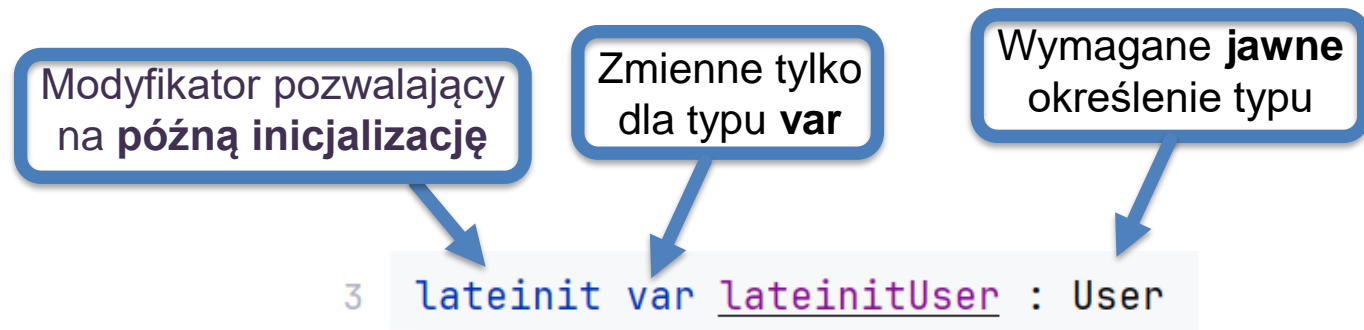
**lateinit** to modyfikator, który pozwala na opóźnioną inicjalizację właściwości zmiennych **tylko dla typu var**.



# Inicjalizacja opóźniona

Gdy nie możemy lub nie chcemy od razu przypisać wartości do zmiennej w momencie jej deklaracji, możemy wykorzystać **opóźnioną inicjalizację**.

**lateinit** to modyfikator, który pozwala na opóźnioną inicjalizację właściwości zmiennych **tylko dla typu var**.



Nie można używać **lateinit** dla **typów prymitywnych**, takich jak **Int** czy **Double**.

```
1 lateinit var name: Int  
[9]
```

at Cell In[9], [line 1](#), column 1: 'lateinit' modifier is not allowed on properties of primitive types

# Inicjalizacja opóźniona

Gdy nie możemy lub nie chcemy od razu przypisać wartości do zmiennej w momencie jej deklaracji, możemy wykorzystać **opóźnioną inicjalizację**.

**lateinit** to modyfikator, który pozwala na opóźnioną inicjalizację właściwości zmiennych **tylko dla typu var**.

Zmienna z **późną  
inicjalizacją**

```
1 class UserView {  
2     lateinit var name: String  
3  
4     fun onCreate() {  
5         name = "Alice"  
6     }  
7  
8     fun printName() {  
9         if (::name.isInitialized) {  
10             println("User's name: $name")  
11         } else {  
12             println("Name is not initialized yet.")  
13         }  
14     }  
15 }
```

# Inicjalizacja opóźniona

Gdy nie możemy lub nie chcemy od razu przypisać wartości do zmiennej w momencie jej deklaracji, możemy wykorzystać **opóźnioną inicjalizację**.

**lateinit** to modyfikator, który pozwala na opóźnioną inicjalizację właściwości zmiennych **tylko dla typu var**.

Zmienna z **późną  
inicjalizacją**

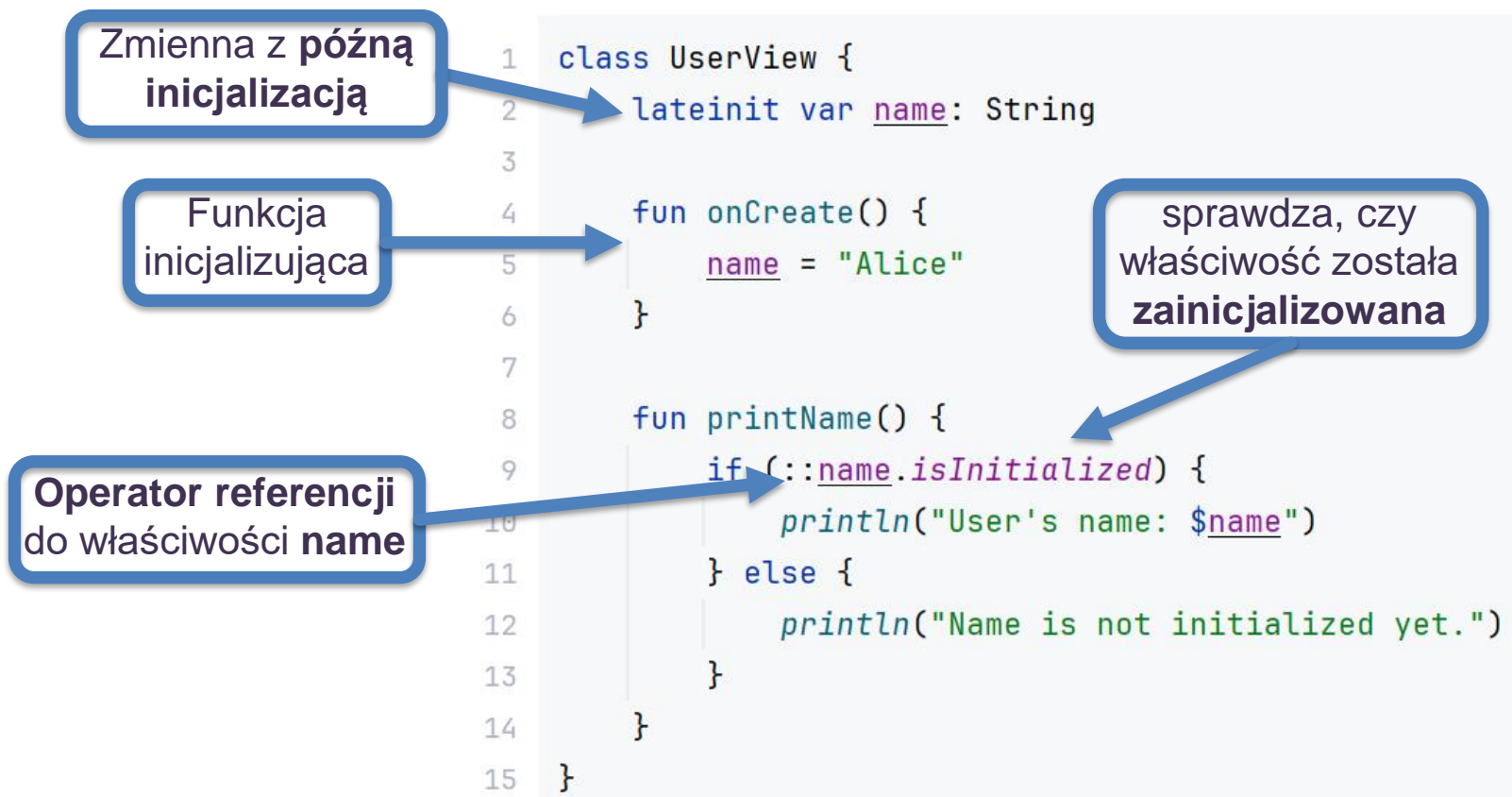
Funkcja  
inicjalizująca

```
1 class UserView {
2     lateinit var name: String
3
4     fun onCreate() {
5         name = "Alice"
6     }
7
8     fun printName() {
9         if (::name.isInitialized) {
10             println("User's name: $name")
11         } else {
12             println("Name is not initialized yet.")
13         }
14     }
15 }
```

# Inicjalizacja opóźniona

Gdy nie możemy lub nie chcemy od razu przypisać wartości do zmiennej w momencie jej deklaracji, możemy wykorzystać **opóźnioną inicjalizację**.

**lateinit** to modyfikator, który pozwala na opóźnioną inicjalizację właściwości zmiennych **tylko dla typu var**.



The diagram illustrates the use of **lateinit** in Kotlin. It shows a class `UserView` with a property `name` declared as `lateinit var name: String`. The `onCreate()` method initializes `name` to "Alice". The `printName()` method checks if `name` is initialized using the `isInitialized` property and prints the name accordingly.

**Zmienna z późną inicjalizacją** points to the `lateinit var name: String` declaration.

**Funkcja inicjalizująca** points to the `onCreate()` method.

**Operator referencji do właściwości name** points to the `::name` property reference in the `if` statement.

**sprawdza, czy właściwość została zainicjalizowana** points to the `isInitialized` property check.

```
1 class UserView {
2     lateinit var name: String
3
4     fun onCreate() {
5         name = "Alice"
6     }
7
8     fun printName() {
9         if (::name.isInitialized) {
10             println("User's name: $name")
11         } else {
12             println("Name is not initialized yet.")
13         }
14     }
15 }
```



**Delegaty** to mechanizm pozwalający na przekazywanie odpowiedzialności za implementację właściwości **innemu obiektowi** - przekazanie wywołania metod **get()** i **set()** do innego obiektu.

Do dyspozycji mamy dwa **wbudowane interfejsy** definiujące sygnatury metod tylko do odczytu oraz do **odczytu i zapisu** właściwości.

nadpisuje jedną z  
wbudowanych operacji

```
1 import kotlin.reflect.KProperty
2
3 interface ReadOnlyProperty<in R, out T> {
4     operator fun getValue(thisRef: R, property: KProperty<*>): T
5 }
6
7 interface ReadWriteProperty<in R, T> {
8     operator fun getValue(thisRef: R, property: KProperty<*>): T
9     operator fun setValue(thisRef: R, property: KProperty<*>, value: T)
10 }
```

**Delegaty** to mechanizm pozwalający na przekazywanie odpowiedzialności za implementację właściwości **innemu obiektowi** - przekazanie wywołania metod **get()** i **set()** do innego obiektu.

Do dyspozycji mamy dwa **wbudowane interfejsy** definiujące sygnatury metod tylko do odczytu oraz do **odczytu i zapisu** właściwości.

parametr typowany może  
być używany **tylko** w  
argumentach funkcji.

nadpisuje jedną z  
**wbudowanych operacji**

```
1 import kotlin.reflect.KProperty
2
3 interface ReadOnlyProperty<in R, out T> {
4     operator fun getValue(thisRef: R, property: KProperty<*>): T
5 }
6
7 interface ReadWriteProperty<in R, T> {
8     operator fun getValue(thisRef: R, property: KProperty<*>): T
9     operator fun setValue(thisRef: R, property: KProperty<*>, value: T)
10 }
```

**Delegaty** to mechanizm pozwalający na przekazywanie odpowiedzialności za implementację właściwości **innemu obiektowi** - przekazanie wywołania metod **get()** i **set()** do innego obiektu.

Do dyspozycji mamy dwa **wbudowane interfejsy** definiujące sygnatury metod tylko do odczytu oraz do **odczytu i zapisu** właściwości.

parametr typowany może  
być używany **tylko** w  
argumentach funkcji.

parametr typowany może  
być używany **tylko** w  
wartościach **zwracanych**  
przez funkcję.

nadpisuje jedną z  
**wbudowanych operacji**

```
1 import kotlin.reflect.KProperty
2
3 interface ReadOnlyProperty<in R, out T> {
4     operator fun getValue(thisRef: R, property: KProperty<*>): T
5 }
6
7 interface ReadWriteProperty<in R, T> {
8     operator fun getValue(thisRef: R, property: KProperty<*>): T
9     operator fun setValue(thisRef: R, property: KProperty<*>, value: T)
10 }
```

**Delegaty** to mechanizm pozwalający na przekazywanie odpowiedzialności za implementację właściwości **innemu obiektowi** - przekazanie wywołania metod **get()** i **set()** do innego obiektu.

Do dyspozycji mamy dwa **wbudowane interfejsy** definiujące sygnatury metod tylko do odczytu oraz do **odczytu i zapisu** właściwości.

parametr typowany może być używany **tylko** w argumentach funkcji.

parametr typowany może być używany **tylko** w wartościach **zwracanych** przez funkcję.

nadpisuje jedną z **wbudowanych operacji**

```
1 import kotlin.reflect.KProperty
2
3 interface ReadOnlyProperty<in R, out T> {
4     operator fun getValue(thisRef: R, property: KProperty<*>): T
5 }
6
7 interface ReadWriteProperty<in R, T> {
8     operator fun getValue(thisRef: R, property: KProperty<*>): T
9     operator fun setValue(thisRef: R, property: KProperty<*>, value: T)
10 }
```

interfejs z biblioteki refleksji, który reprezentuje **właściwość klasy** lub **obiektu**. Pozwala na **dostęp do metadanych** właściwości

# Delegaty

**Delegaty** to mechanizm pozwalający na przekazywanie odpowiedzialności za implementację właściwości **innemu obiektowi** - przekazanie wywołania metod **get()** i **set()** do innego obiektu.

Do dyspozycji mamy dwa **wbudowane interfejsy** definiujące sygnatury metod tylko do odczytu oraz do **odczytu i zapisu** właściwości.

parametr typowany może być używany **tylko** w argumentach funkcji.

parametr typowany może być używany **tylko** w wartościach **zwracanych** przez funkcję.

nadpisuje jedną z **wbudowanych operacji**

```
1 import kotlin.reflect.KProperty
2
3 interface ReadOnlyProperty<in R, out T> {
4     operator fun getValue(thisRef: R, property: KProperty<*>): T
5 }
6
7 interface ReadWriteProperty<in R, T> {
8     operator fun getValue(thisRef: R, property: KProperty<*>): T
9     operator fun setValue(thisRef: R, property: KProperty<*>, value: T)
10 }
```

interfejs z biblioteki refleksji, który reprezentuje **właściwość klasy** lub **obiektu**. Pozwala na **dostęp do metadanych** właściwości

**Typ dziki**, może być **dowolnym typem**, ale **bez narzucania** jakiegokolwiek **hierarchii typów**

**Delegat** jest obsługiwany za pomocą słowa kluczowego **by**, które wskazuje, że właściwość jest **delegowana do konkretnej implementacji**.

Klasa **implementuje interfejs**  
umożliwiający nadpisanie  
**gettera i settera**

```
4 class UpperCaseDelegate : ReadWriteProperty<Any?, String> {  
5     private var value: String = ""  
6  
7     override fun getValue(thisRef: Any?, property: KProperty<*>): String {  
8         return value  
9     }  
10  
11     override fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {  
12         this.value = value.uppercase()  
13     }  
14 }
```

**Delegat** jest obsługiwany za pomocą słowa kluczowego **by**, które wskazuje, że właściwość jest **delegowana do konkretnej implementacji**.

Klasa implementuje interfejs  
umożliwiający nadpisanie  
**gettera i settera**

Klasa musi  
nadpisać  
**wszystkie**  
metody

```
4 class UpperCaseDelegate : ReadWriteProperty<Any?, String> {  
5     private var value: String = ""  
6  
7     override fun getValue(thisRef: Any?, property: KProperty<*>): String {  
8         return value  
9     }  
10  
11     override fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {  
12         this.value = value.uppercase()  
13     }  
14 }
```

**Delegat** jest obsługiwany za pomocą słowa kluczowego **by**, które wskazuje, że właściwość jest **delegowana do konkretnej implementacji**.

Klasa **implementuje interfejs**  
umożliwiający nadpisanie  
**gettera i settera**

Klasa **musi**  
nadpisać  
**wszystkie**  
metody

```
4 class UpperCaseDelegate : ReadWriteProperty<Any?, String> {  
5     private var value: String = ""  
6  
7     override fun getValue(thisRef: Any?, property: KProperty<*>): String {  
8         return value  
9     }  
10  
11     override fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {  
12         this.value = value.uppercase()  
13     }  
14 }
```

**Brak słowa kluczowego**  
**operator**



**Delegat** jest obsługiwany za pomocą słowa kluczowego **by**, które wskazuje, że właściwość jest **delegowana do konkretnej implementacji**.

Klasa implementuje interfejs  
umożliwiający nadpisanie  
**gettera i settera**

Klasa musi  
nadpisać  
**wszystkie**  
metody

```
4 class UpperCaseDelegate : ReadWriteProperty<Any?, String> {  
5     private var value: String = ""  
6  
7     override fun getValue(thisRef: Any?, property: KProperty<*>): String {  
8         return value  
9     }  
10  
11     override fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {  
12         this.value = value.uppercase()  
13     }  
14 }
```

Brak słowa kluczowego  
**operator**

Deleguje właściwość -przenosi  
odpowiedzialność za **zarządzanie** wartością  
właściwości do **innego obiektu**

```
1 var userName: String by UpperCaseDelegate()  
2  
3 userName = "kotlin"  
4 println(userName)
```

**Delegat** jest obsługiwany za pomocą słowa kluczowego **by**, które wskazuje, że właściwość jest **delegowana do konkretnej implementacji**.

Klasa implementuje interfejs  
umożliwiający nadpisanie  
**gettera i settera**

Klasa musi  
nadpisać  
**wszystkie**  
metody

```
4 class UpperCaseDelegate : ReadWriteProperty<Any?, String> {  
5     private var value: String = ""  
6  
7     override fun getValue(thisRef: Any?, property: KProperty<*>): String {  
8         return value  
9     }  
10  
11     override fun setValue(thisRef: Any?, property: KProperty<*>, value: String) {  
12         this.value = value.uppercase()  
13     }  
14 }
```

Brak słowa kluczowego  
**operator**

Deleguje właściwość -przenosi  
odpowiedzialność za **zarządzanie** wartością  
właściwości do **innego obiektu**


```
1 var userName: String by UpperCaseDelegate()  
2  
3 userName = "kotlin"  
4 println(userName)
```

**Obiekt delegujący** - zawiera logikę, która  
modyfikuje wartość właściwości przed jej  
ustawieniem lub po jej odczycie

# Delegat z obserwatorem

Jednym z praktycznych zastosowań delegatów jest tworzenie **mechanizmu obserwacji zmian właściwości**. Dzięki temu możemy wykonać określone działania **za każdym razem**, gdy właściwość **zmienia swoją wartość**.

**Delegowanie właściwości**  
do innego obiektu



```
3 var userName: String by Delegates.observable("DefaultUser") { property, oldValue, newValue ->
4     println("Właściwość ${property.name} zmieniona z \"$oldValue\" na \"$newValue\"")
5 }
```

# Delegat z obserwatorem

Jednym z praktycznych zastosowań delegatów jest tworzenie **mechanizmu obserwacji zmian właściwości**. Dzięki temu możemy wykonać określone działania **za każdym razem**, gdy właściwość **zmienia swoją wartość**.

Delegowanie właściwości  
do innego obiektu

delegat **observable** służy  
do śledzenia **zmian**  
**wartości właściwości**

Wartość **domyślna**

```
3 var userName: String by Delegates.observable("DefaultUser") { property, oldValue, newValue ->
4   println("Właściwość ${property.name} zmieniona z \"$oldValue\" na \"$newValue\"")
5 }
```

# Delegat z obserwatorem

Jednym z praktycznych zastosowań delegatów jest tworzenie **mechanizmu obserwacji zmian właściwości**. Dzięki temu możemy wykonać określone działania **za każdym razem**, gdy właściwość **zmienia swoją wartość**.

Delegowanie właściwości  
do innego obiektu

delegat **observable** służy  
do śledzenia **zmian**  
wartości właściwości

Wartość **domyślna**

```
3 var userName: String by Delegates.observable("DefaultUser") { property, oldValue, newValue ->
4   println("Właściwość ${property.name} zmieniona z \"$oldValue\" na \"$newValue\"")
5 }
```

Lambda wywołana przy  
każdej zmianie  
właściwości

# Delegat z obserwatorem

Jednym z praktycznych zastosowań delegatów jest tworzenie **mechanizmu obserwacji zmian właściwości**. Dzięki temu możemy wykonać określone działania **za każdym razem**, gdy właściwość **zmienia swoją wartość**.

Delegowanie właściwości  
do innego obiektu

delegat **observable** służy  
do śledzenia **zmian**  
**wartości właściwości**

Wartość **domyślna**

```
3 var userName: String by Delegates.observable("DefaultUser") { property, oldValue, newValue ->
4   println("Właściwość ${property.name} zmieniona z \"$oldValue\" na \"$newValue\"")
5 }
```

Lambda wywołana przy  
**każdej zmianie**  
właściwości

Obiekt reprezentujący właściwość, która  
została zmieniona (w tym przypadku  
**userName**). Możemy uzyskać dostęp  
np. do jej **nazwy**.

# Delegat z obserwatorem

Jednym z praktycznych zastosowań delegatów jest tworzenie **mechanizmu obserwacji zmian właściwości**. Dzięki temu możemy wykonać określone działania **za każdym razem**, gdy właściwość **zmienia swoją wartość**.

Delegowanie właściwości  
do innego obiektu

delegat **observable** służy  
do śledzenia **zmian**  
**wartości właściwości**

Wartość **domyślna**

```
3 var userName: String by Delegates.observable("DefaultUser") { property, oldValue, newValue ->
4   println("Właściwość ${property.name} zmieniona z \"$oldValue\" na \"$newValue\"")
5 }
```

Lambda wywołana przy  
**każdej zmianie**  
właściwości

**Poprzednia** wartość  
właściwości przed jej  
zmianą.

Obiekt reprezentujący właściwość, która  
została zmieniona (w tym przypadku  
**userName**). Możemy uzyskać dostęp  
np. do jej **nazwy**.

**Nowa wartość**, którą  
przypisano do  
właściwości.

# Delegat z obserwatorem

Jednym z praktycznych zastosowań delegatów jest tworzenie **mechanizmu obserwacji zmian właściwości**. Dzięki temu możemy wykonać określone działania **za każdym razem**, gdy właściwość **zmienia swoją wartość**.

Delegowanie właściwości  
do innego obiektu

delegat **observable** służy  
do śledzenia **zmian**  
**wartości właściwości**

Wartość domyślna

```
3 var userName: String by Delegates.observable("DefaultUser") { property, oldValue, newValue ->
4   println("Właściwość ${property.name} zmieniona z \"$oldValue\" na \"$newValue\"")
5 }
```

Przy każdej zmianie  
wykonywana jest funkcja  
lambda

Nazwa właściwości

Wartości właściwości

```
7 userName = "Alice"
8 userName = "Bob"
[7]
```

Właściwość userName zmieniona z "DefaultUser" na "Alice"

Właściwość userName zmieniona z "Alice" na "Bob"



# Delegat wetowalny

Delegat **Delegates.vetoable** umożliwia kontrolowanie **zmiany wartości** właściwości. Każda zmiana jest **poprzedzona wywołaniem bloku kodu**, w którym możemy sprawdzić, czy nowa wartość spełnia określone warunki.

```
3 var max: Int by Delegates.vetoable(0) { _, oldValue, newValue ->
4     if (newValue > oldValue)
5         true ^vetoable
6     else {
7         println("New value must be larger than old value.")
8         false ^vetoable
9     }
10 }
```

# Delegat wetowalny

Delegat **Delegates.vetoable** umożliwia kontrolowanie **zmiany wartości** właściwości. Każda zmiana jest **poprzedzona wywołaniem bloku kodu**, w którym możemy sprawdzić, czy nowa wartość spełnia określone warunki.

Delegowanie właściwości  
do innego obiektu

Delegat **vetoable**  
kontroluje **zmiany  
wartości**

Wartość domyślna

```
3 var max: Int by Delegates.vetoable(0) { _, oldValue, newValue ->
4   if (newValue > oldValue)
5     true ^vetoable
6   else {
7     println("New value must be larger than old value.")
8     false ^vetoable
9   }
10 }
```

# Delegat wetowalny

Delegat **Delegates.vetoable** umożliwia kontrolowanie **zmiany wartości** właściwości. Każda zmiana jest **poprzedzona wywołaniem bloku kodu**, w którym możemy sprawdzić, czy nowa wartość spełnia określone warunki.

Delegowanie właściwości  
do innego obiektu

Delegat **vetoable**  
kontroluje **zmiany  
wartości**

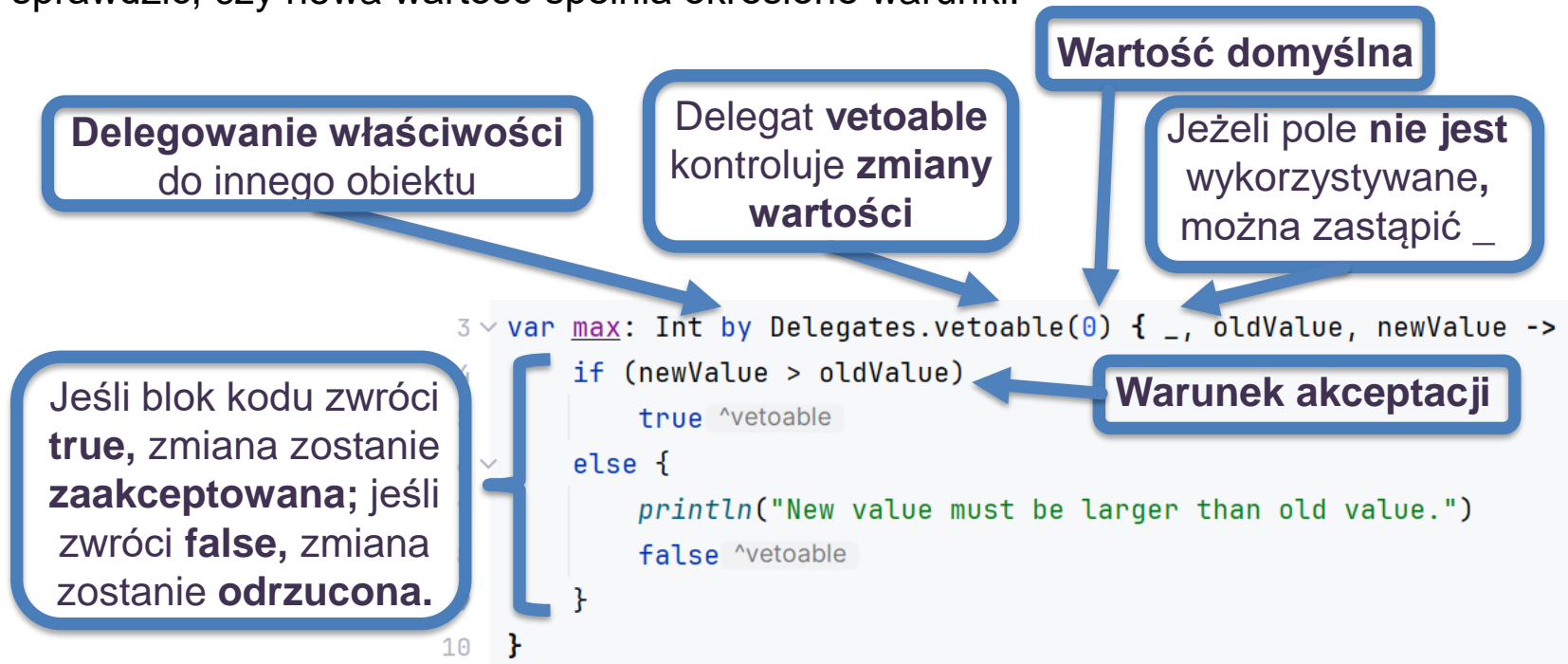
Wartość domyślna

Jeżeli pole **nie jest**  
wykorzystywane,  
można zastąpić **\_**

```
3 var max: Int by Delegates.vetoable(0) { _, oldValue, newValue ->
4   if (newValue > oldValue)
5     true ^vetoable
6   else {
7     println("New value must be larger than old value.")
8     false ^vetoable
9   }
10 }
```

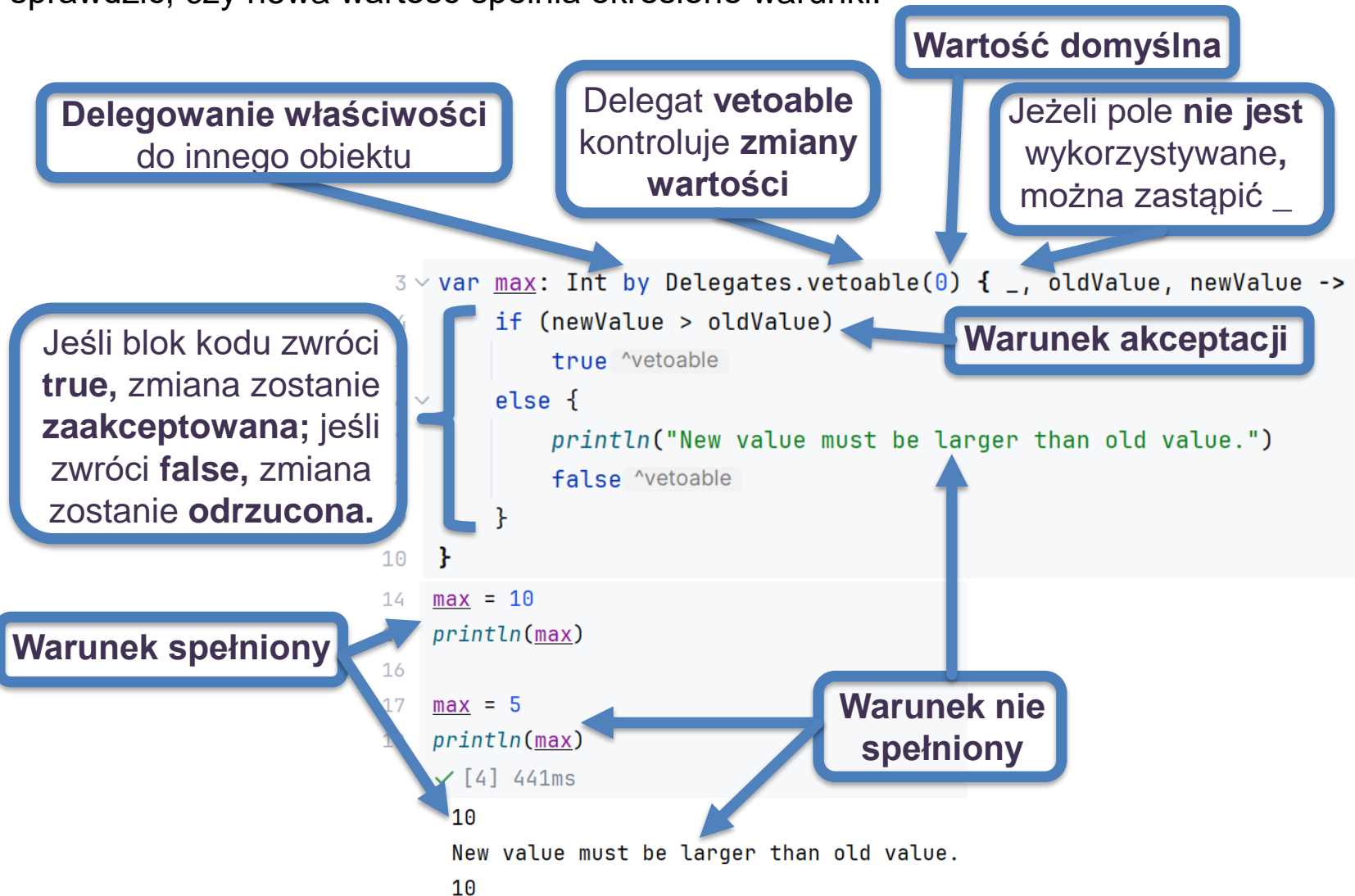
# Delegat wetowalny

Delegat **Delegates.vetoable** umożliwia kontrolowanie **zmiany wartości** właściwości. Każda zmiana jest **poprzedzona wywołaniem bloku kodu**, w którym możemy sprawdzić, czy nowa wartość spełnia określone warunki.



# Delegat wetowalny


Delegat **Delegates.vetoable** umożliwia kontrolowanie **zmiany wartości** właściwości. Każda zmiana jest **poprzedzona wywołaniem bloku kodu**, w którym możemy sprawdzić, czy nowa wartość spełnia określone warunki.



# Delegat do mapy

W delegacie **do mapy** wartości właściwości klasy są **pobierane z mapy** na podstawie ich **nazw jako kluczy**.

Konstruktor główny posiada  
jedno pole typu  
**Map<String, Any?>**



```
1 data class User(val myMap: Map<String, Any?>) {  
2     val name: String by myMap  
3     val age: Int by myMap  
4 }  
5  
6 val user = User(mapOf(  
7     "name" to "Rafał Lewandków",  
8     "age" to 30  
9 ))
```

# Delegat do mapy

W delegacie **do mapy** wartości właściwości klasy są **pobierane z mapy** na podstawie ich **nazw jako kluczy**.

Konstruktor główny posiada  
jedno pole typu  
**Map<String, Any?>**

właściwości zostaną pobrane  
z mapy. Mapa używana jako  
źródło danych **musi zawierać  
klucze zgodne z nazwami  
właściwości**.

```
1 data class User(val myMap: Map<String, Any?>) {  
2     val name: String by myMap  
3     val age: Int      by myMap  
4 }  
5  
6 val user = User(mapOf(  
7     "name" to "Rafał Lewandków",  
8     "age"  to 30  
9 ))
```

# Inicjalizacja leniwa

**Inicjalizacja leniwa** to technika, która pozwala na **opóźnienie zainicjalizowania** właściwości aż do momentu, gdy zostanie ona **po raz pierwszy użyta**. Realizuje się to za pomocą delegatu **lazy**.

Delegowanie właściwości  
do innego obiektu

Inicjalizacja leniwa

```
1 val lazyValue: String by lazy {  
2     println("Obliczanie wartości...")  
3     "Hello, Kotlin!" ^lazy  
4 }  
5  
6 println(lazyValue)  
7 println(lazyValue)  
[15]
```

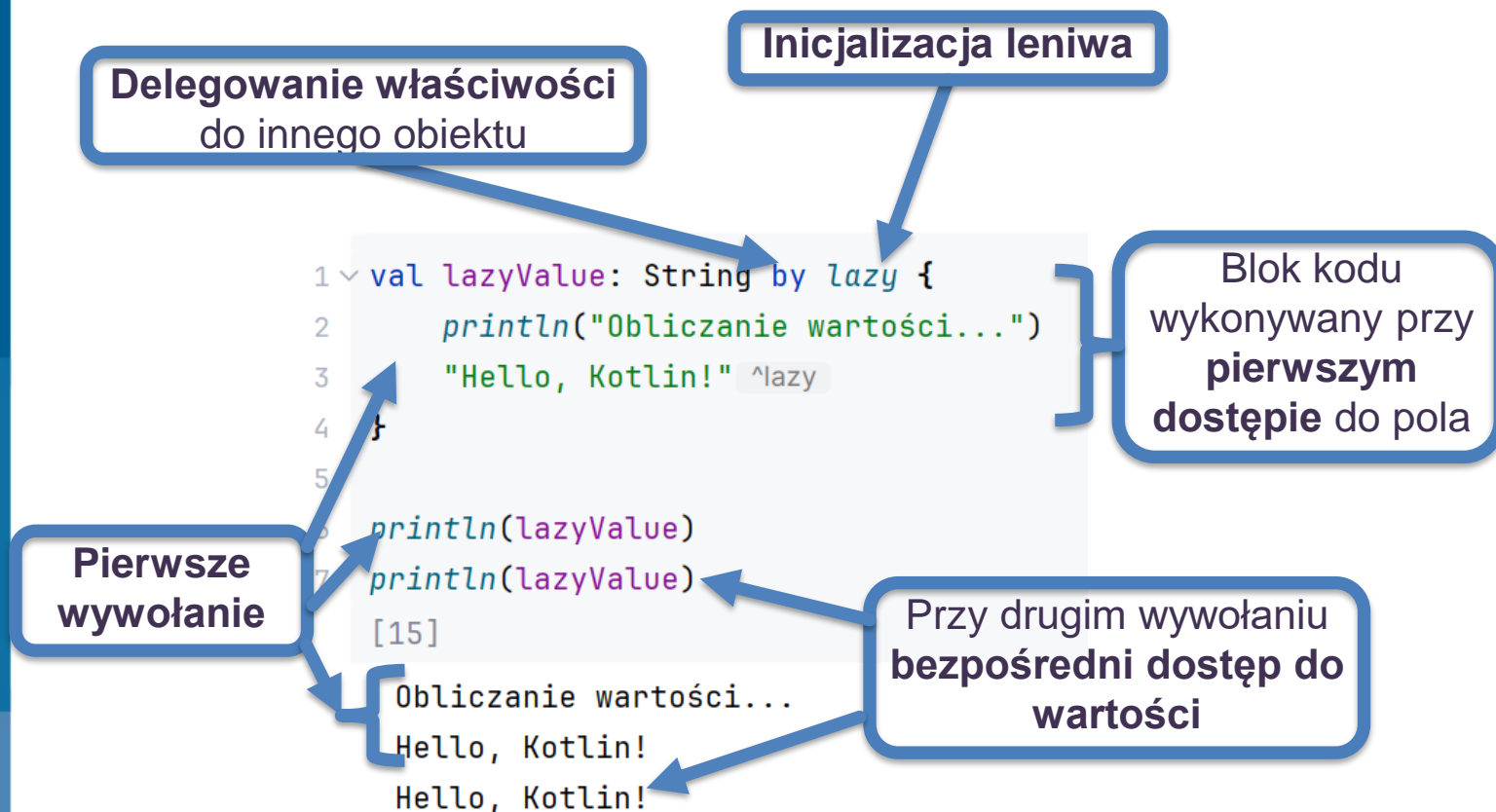
Blok kodu  
wykonywany przy  
**pierwszym**  
**dostępie** do pola

```
Obliczanie wartości...  
Hello, Kotlin!  
Hello, Kotlin!
```



# Inicjalizacja leniwa

**Inicjalizacja leniwa** to technika, która pozwala na **opóźnienie zainicjalizowania** właściwości aż do momentu, gdy zostanie ona **po raz pierwszy użyta**. Realizuje się to za pomocą delegatu **lazy**.



**lateinit** i **lazy** to **mechanizmy opóźnionej inicjalizacji** w Kotlinie, które różnią się w kontekście ich użycia, typów danych oraz mechaniki działania.

- **lateinit:**

- Jest używane do **deklaracji właściwości** które nie są inicjalizowane w momencie ich stworzenia.
- **Nie można ich stosować z typami prymitywnymi.**
- Zmienne te muszą być zadeklarowane jako **var**.
- Brak inicjalizacji przed pierwszym użyciem wywoła wyjątek **UninitializedPropertyAccessException**.
- Właściwości nie mogą być deklarowane jako **nullable**.

- **lazy**

- Służy do inicjalizacji właściwości **val**.
- Inicjalizacja następuje dopiero w momencie pierwszego dostępu do właściwości.
- Działa na **wszystkich typach danych**, w tym również na **typach prymitywnych**.
- właściwość może być zarówno **nullable** (**String?**), jak i **non-nullable** (**String**).

# Delegat nullable()

Jeżeli zachodzi potrzeba **późnej inicjalizacji** zmiennej (**var**) z **typem prymitywnym**, możemy wykorzystać delegat **nullable**. Zapobiega przypisaniu wartości **null** do właściwości.

Delegowanie właściwości  
do innego obiektu

Wymaga, aby przed pierwszym  
użyciem przypisać do zmiennej  
faktyczną wartość.

```
3 var myInt: Int by Delegates.notNull()
4
5 myInt = 2
6 println(myInt)
7
8 myInt = 10
9 println(myInt)
[24]
2
10
```