



PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 1

WYKŁAD 5

- Klasy

Klasa służy jako **szablon** do wytwarzania obiektów. Obiekty są **instancjami** klasy, które łączą w sobie dane oraz funkcjonalność.

```
1  class Points {  
2      var points = 0  
3      val max = 10  
4  
5      fun add(increase: Int): Int {  
6          points += increase  
7          if (points > max)  
8              points = max  
9          return points  
10     }  
11 }  
[1]
```

Klasa służy jako **szablon** do wytwarzania obiektów. Obiekty są **instancjami** klasy, które łączą w sobie dane oraz funkcjonalność.

```
1 class Points {  
2     var points = 0  
3     val max = 10  
4  
5     fun add(increase: Int): Int {  
6         points += increase  
7         if (points > max)  
8             points = max  
9         return points  
10    }  
11 }  
[1]
```

Stan obiektu reprezentowany jest poprzez **właściwości**, które są **zmiennymi** lub **wartościami** zdefiniowanymi **wewnątrz klasy**.

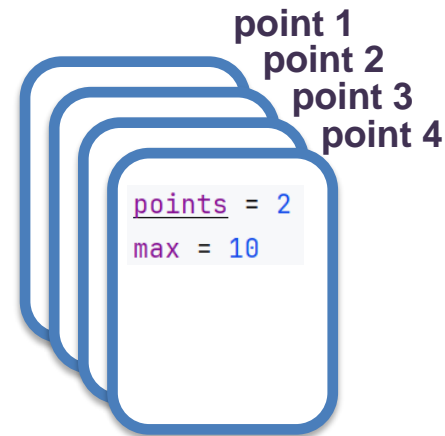
Metody definiują **zachowanie klasy**, czyli operacje, które można wykonać na obiektach tej klasy.

Klasa służy jako **szablon** do wytwarzania obiektów. Obiekty są **instancjami** klasy, które łączą w sobie dane oraz funkcjonalność.

```
1 class Points {  
2     var points = 0  
3     val max = 10  
4  
5     fun add(increase: Int): Int {  
6         points += increase  
7         if (points > max)  
8             points = max  
9         return points  
10    }  
11 }  
[1]
```

Stan obiektu reprezentowany jest poprzez **właściwości**, które są **zmiennymi** lub **wartościami** zdefiniowanymi **wewnątrz klasy**.

Metody definiują **zachowanie klasy**, czyli operacje, które można wykonać na obiektach tej klasy.

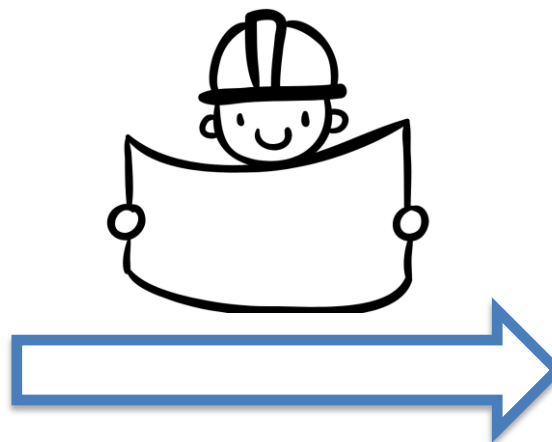


Konstruktor to specjalna funkcja, która służy do **inicjalizacji obiektów** danej klasy. Jest wywoływana **automatycznie** w momencie tworzenia nowego obiektu i pozwala na ustawienie początkowego stanu właściwości klasy.

- **Konstruktor główny** – jest zintegrowany z definicją klasy.
- **Konstruktor drugorzędny** – jest definiowany jako osobna funkcja w ciele klasy

Klasa

```
1 class Points {  
2     var points = 0  
3     val max = 10  
4  
5     fun add(increase: Int): Int {  
6         points += increase  
7         if (points > max)  
8             points = max  
9         return points  
10    }  
11 }  
[1]
```



Instancje klasy

```
val point1 = Points()  
val point2 = Points()  
val point3 = Points()  
val point4 = Points()
```



Konstruktor główny

Konstruktor główny jest definiowany bezpośrednio po nazwie klasy w nawiasach okrągłych. Może przyjmować parametry, które można wykorzystać do inicjalizacji właściwości klasy.

```
class Student constructor(val firstName: String, var index: Int) {}
```

Pozwala na deklarowanie parametrów jako **val** lub **var**, ponieważ parametry te są **automatycznie** traktowane jako **właściwości klasy**.

Konstruktor główny jest definiowany bezpośrednio po nazwie klasy w nawiasach okrągłych. Może przyjmować parametry, które można wykorzystać do inicjalizacji właściwości klasy.

słowo
kluczowe

nazwa
klasy

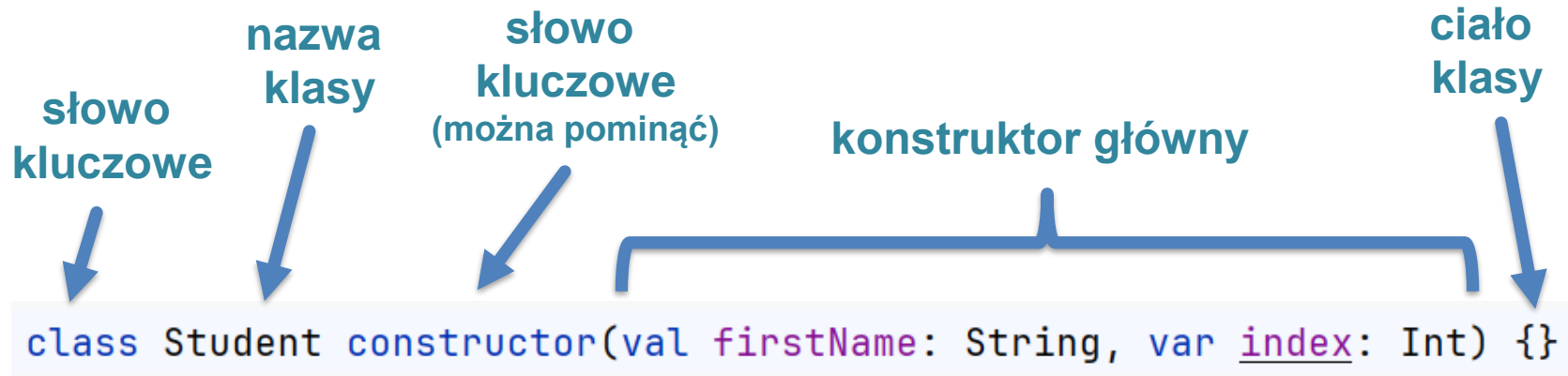
słowo
kluczowe
(można pominąć)

```
class Student constructor(val firstName: String, var index: Int) {}
```

Pozwala na deklarowanie parametrów jako **val** lub **var**, ponieważ parametry te są **automatycznie** traktowane jako **właściwości klasy**.

Konstruktor główny

Konstruktor główny jest definiowany bezpośrednio po nazwie klasy w nawiasach okrągłych. Może przyjmować parametry, które można wykorzystać do inicjalizacji właściwości klasy.

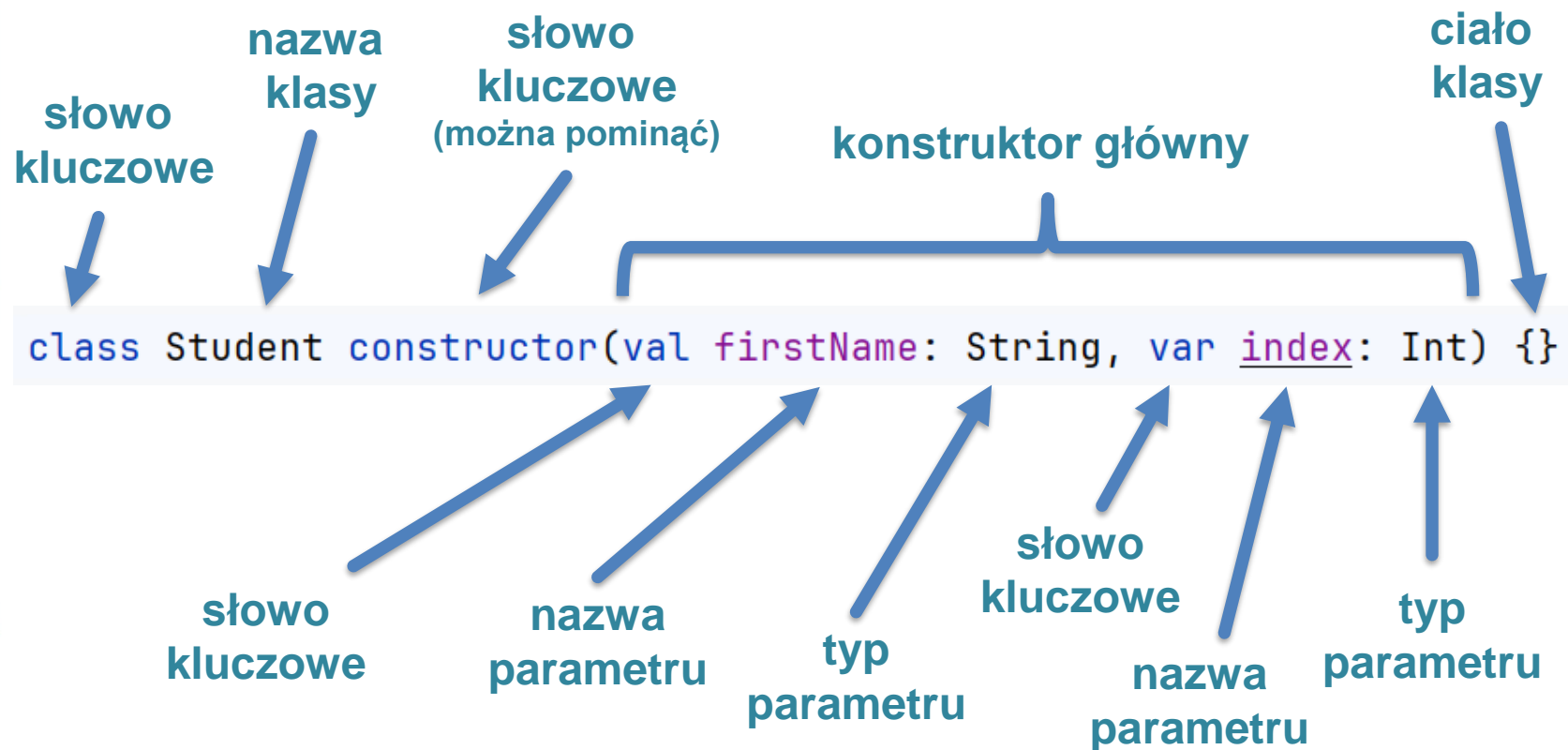


```
class Student constructor(val firstName: String, var index: Int) {}
```

Pozwala na deklarowanie parametrów jako **val** lub **var**, ponieważ parametry te są **automatycznie** traktowane jako **właściwości klasy**.

Konstruktor główny

Konstruktor główny jest definiowany bezpośrednio po nazwie klasy w nawiasach okrągłych. Może przyjmować parametry, które można wykorzystać do inicjalizacji właściwości klasy.



Pozwala na deklarowanie parametrów jako **val** lub **var**, ponieważ parametry te są **automatycznie** traktowane jako **właściwości klasy**.

Konstruktor główny nie może zawierać kodu, **kod inicjacyjny** może zostać umieszczony w bloku **init**.

```
1  class Student (val name: String, var index: Int) {  
2  
3      // Initializer Block  
4      init {  
5          println("Name = $name")  
6          println("Index number = $index")  
7      }  
8  }
```

Klasa może posiadać wiele bloków **init**, są one wykonywane w kolejności umieszczenia w klasie i wywołane w momencie wywołania konstruktora głównego.

Konstruktor główny

Konstruktor główny nie może zawierać kodu, **kod inicjacyjny** może zostać umieszczony w bloku **init**.

```
1 class Student (val name: String, var index: Int) {  
2  
3     // Initializer Block  
4     init {  
5         println("Name = $name")  
6         println("Index number = $index")  
7     }  
8 }
```

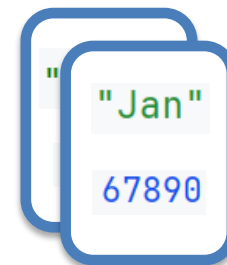
Klasa może posiadać wiele bloków **init**, są one wykonywane w kolejności umieszczenia w klasie i wywołane w momencie wywołania konstruktora głównego.

Klasa

```
1 class Student (val name: String, var index: Int) {  
2  
3     // Initializer Block  
4     init {  
5         println("Name = $name")  
6         println("Index number = $index")  
7     }  
8 }
```



Instancje klasy



```
1 // Tworzenie obiektu klasy Student  
2 val student1 = Student("Anna", 12345)  
3 val student2 = Student("Jan", 67890)
```

Tworzenie obiektów na podstawie szablonu klasy polega na wywołaniu jej konstruktora.

```
1 // Tworzenie obiektu klasy Student
2 val student1 = Student("Anna", 12345)
3 val student2 = Student("Jan", 67890)
✓ [6] 281ms
```

Name = Anna

Index number = 12345

Name = Jan

Index number = 67890

Automatycznie wywołany
blok **init** przy wywołaniu
konstruktora głównego

Konstruktor główny

Tworzenie obiektów na podstawie szablonu klasy polega na wywołaniu jej konstruktora.

```
1 // Tworzenie obiektu klasy Student
2 val student1 = Student("Anna", 12345)
3 val student2 = Student("Jan", 67890)
✓ [6] 281ms
```

Name = Anna

Index number = 12345

Name = Jan

Index number = 67890

Automatycznie wywołany
blok **init** przy wywołaniu
konstruktora głównego

```
1 // Możemy uzyskać dostęp do właściwości obiektu
2 println(student1.name)
3 println(student1.index)
✓ [7] 111ms
```

Anna

12345

```
1 class Student (val name: String, var index: Int) {
2
3     // Initializer Block
4     init {
5         println("Name = $name")
6         println("Index number = $index")
7     }
8 }
```

Konstruktor drugorzędny

- Przy tworzeniu **drugorzędnych konstruktorów** wymagane jest użycie słowa kluczowego **constructor**.
- Jest bardziej elastyczny niż konstruktor główny, **może posiadać ciało**.
- **Nie ma możliwości automatycznego** deklarowania właściwości za pomocą **val** lub **var**.
- Parametry konstruktora drugorzędnego są traktowane wyłącznie jako **lokalne zmienne** dostępne w jego ciele.

```
1 class Student{
2     var name: String
3     var index: Int
4
5     // constructor (val _name: String, var _index: Int)
6     constructor (_name: String, _index: Int) {
7         this.name = _name
8         this.index = _index
9         println("Name = $name")
10        println("Index number = $index")
11    }
12 }
```

Konstruktor drugorzędny

Każda klasa wystawia interfejs (konstruktory, metody, pola) który jest dostępny z zewnątrz klasy.

- **public** - pole, metoda lub konstruktor z modyfikatorem publicznym, jest dostępne z każdego miejsca - **modyfikator domyślny**
- **protected** - pola, metody i konstruktory klasy oznaczone w ten sposób są dostępne w obrębie klasy oraz we wszystkich podklasach.
- **private** - do pól, metod i konstruktorów prywatnych nie można uzyskać dostępu spoza klasy w której się znajdują
- **internal** - obiekty widoczne w obrębie modułu lub pakietu

```
1  class BankAccount(private val accountNumber: String, internal val balance: Double) {  
2  
3      // Public method - dostępna wszędzie  
4      fun displayAccountInfo() {  
5          println("Account Number: $accountNumber")  
6          println("Balance: $balance")  
7      }  
8  
9      // Private method - dostępna tylko w obrębie klasy  
10     private fun calculateInterest(): Double {  
11         return balance * 0.05  
12     }  
13 }
```

Obiekt stowarzyszony (**companion object**) to sposób na tworzenie **statycznych elementów** w klasach.

```
1 class MyClass {  
2     companion object {  
3         const val CONSTANT = "Stała wartość"  
4         fun staticMethod() {  
5             println("Wywołano metodę statyczną!")  
6         }  
7     }  
8 }
```

Elementy zadeklarowane w obiekcie stowarzyszonym można wywoływać **bez tworzenia instancji klasy**.

Obiekt stowarzyszony (**companion object**) to sposób na tworzenie **statycznych elementów** w klasach.

```
1 class MyClass {  
2     companion object {  
3         const val CONSTANT = "Stała wartość"  
4         fun staticMethod() {  
5             println("Wywołano metodę statyczną!")  
6         }  
7     }  
8 }
```

Elementy zadeklarowane w obiekcie stowarzyszonym można wywoływać **bez tworzenia instancji klasy**.

```
1 println(MyClass.CONSTANT)  
2 MyClass.staticMethod()  
✓ [14] 92ms
```

Stała wartość

Wywołano metodę statyczną!

Obiekt stowarzyszony (**companion object**) to sposób na tworzenie **statycznych elementów** w klasach.

```
1 class MyClass {  
2     companion object {  
3         const val CONSTANT = "Stała wartość"  
4         fun staticMethod() {  
5             println("Wywołano metodę statyczną!")  
6         }  
7     }  
8 }
```

Elementy zadeklarowane w obiekcie stowarzyszonym można wywoływać **bez tworzenia instancji klasy**.

```
1 println(MyClass.CONSTANT)  
2 MyClass.staticMethod()  
✓ [14] 92ms
```

Stała wartość

Wywołano metodę statyczną!

- Elementy companion object są udostępniane na **poziomie klasy**, a nie instancji.
- Companion object jest **pojedynczą instancją** (*singletonem*) powiązaną z klasą.
- Może implementować interfejsy.
- Może posiadać nazwę.

Porównanie obiektów

W Kotlinie porównywanie obiektów można przeprowadzać na dwa sposoby: **porównanie strukturalne** (operator `==`) i **porównanie referencyjne** (operator `===`).

- Operator `==` porównuje wartości zmiennych.
- Operator `===` sprawdza, czy zmienne wskazują na tę samą lokalizację w pamięci.

```
1 class Person(val name: String)
2 val p1 = Person("Anna")
3 val p2 = Person("Karol")
```

Pamięć

| | | | | |
|-----------------|--|--|------------------|--|
| Anna 0x09ead | | | | |
| | | | Karol 0xaa16b | |

Porównanie obiektów

W Kotlinie porównywanie obiektów można przeprowadzać na dwa sposoby: **porównanie strukturalne** (operator `==`) i **porównanie referencyjne** (operator `===`).

- Operator `==` porównuje wartości zmiennych.
- Operator `===` sprawdza, czy zmienne wskazują na tę samą lokalizację w pamięci.

```
1 class Person(val name: String)
2 val p1 = Person("Anna")
3 val p2 = Person("Karol")
```

porównanie strukturalne

```
println(p1 == p2)
```

Pamięć

| | | | | |
|-----------------|--|--|------------------|--|
| Anna 0x09ead | | | | |
| | | | Karol 0xaa16b | |

Porównanie obiektów

W Kotlinie porównywanie obiektów można przeprowadzać na dwa sposoby: **porównanie strukturalne** (operator `==`) i **porównanie referencyjne** (operator `===`).

- Operator `==` porównuje wartości zmiennych.
- Operator `===` sprawdza, czy zmienne wskazują na tę samą lokalizację w pamięci.

```
1 class Person(val name: String)
2 val p1 = Person("Anna")
3 val p2 = Person("Karol")
```

porównanie strukturalne

```
println(p1 == p2)
```

```
"Anna" == "Karol"
```

false

Pamięć

| | | | | |
|-----------------|--|--|------------------|--|
| Anna 0x09ead | | | | |
| | | | Karol 0xaa16b | |

Porównanie obiektów

W Kotlinie porównywanie obiektów można przeprowadzać na dwa sposoby: **porównanie strukturalne** (operator `==`) i **porównanie referencyjne** (operator `===`).

- Operator `==` porównuje wartości zmiennych.
- Operator `===` sprawdza, czy zmienne wskazują na tę samą lokalizację w pamięci.

```
1 class Person(val name: String)
2 val p1 = Person("Anna")
3 val p2 = Person("Karol")
```

porównanie referencyjne

```
println(p1 === p2)
```

```
0x09ead === 0xaa16b
```

false

Pamięć

| | | | | |
|-----------------|--|--|------------------|--|
| Anna 0x09ead | | | | |
| | | | Karol 0xaa16b | |

Porównanie obiektów

W Kotlinie porównywanie obiektów można przeprowadzać na dwa sposoby: **porównanie strukturalne** (operator `==`) i **porównanie referencyjne** (operator `===`).

- Operator `==` porównuje wartości zmiennych.
- Operator `===` sprawdza, czy zmienne wskazują na tę samą lokalizację w pamięci.

```
1 class Person(val name: String)
2 val p1 = Person("Anna")
3 val p2 = Person("Karol")
```

porównanie referencyjne

```
println(p1 === p2)
```

```
0x09ead === 0xaa16b
```

false

Pamięć

| | | | | |
|-----------------|--|--|------------------|--|
| Anna 0x09ead | | | | |
| | | | Karol 0xaa16b | |

Porównanie obiektów

W Kotlinie porównywanie obiektów można przeprowadzać na dwa sposoby: **porównanie strukturalne** (operator `==`) i **porównanie referencyjne** (operator `===`).

- Operator `==` porównuje wartości zmiennych.
- Operator `===` sprawdza, czy zmienne wskazują na tę samą lokalizację w pamięci.

```
1 class Person(val name: String)
2 val p1 = Person("Anna")
3 val p2 = Person("Anna")
```

porównanie strukturalne

```
println(p1 == p2)
"Anna" == "Anna"
```

Pamięć

| | | | | |
|-----------------|--|--|-----------------|--|
| Anna 0x09ead | | | | |
| | | | Anna 0xaa16b | |

Porównanie obiektów

W Kotlinie porównywanie obiektów można przeprowadzać na dwa sposoby: **porównanie strukturalne** (operator `==`) i **porównanie referencyjne** (operator `===`).

- Operator `==` porównuje wartości zmiennych.
- Operator `===` sprawdza, czy zmienne wskazują na tę samą lokalizację w pamięci.

```
1 class Person(val name: String)
2 val p1 = Person("Anna")
3 val p2 = Person("Anna")
```

porównanie strukturalne

```
println(p1 == p2)
```

```
"Anna" == "Anna"
```

false

Pamięć

| | | | | |
|-----------------|--|--|-----------------|--|
| Anna 0x09ead | | | | |
| | | | Anna 0xaa16b | |

Porównanie obiektów

W Kotlinie porównywanie obiektów można przeprowadzać na dwa sposoby: **porównanie strukturalne** (operator `==`) i **porównanie referencyjne** (operator `===`).

- Operator `==` porównuje wartości zmiennych.
- Operator `===` sprawdza, czy zmienne wskazują na tę samą lokalizację w pamięci.

```
1 class Person(val name: String)
2 val p1 = Person("Anna")
3 val p2 = Person("Anna")
4
5 println(p1 == p2)
6 println(p1 === p2)
```

✓ [18] 99ms

false

false

Przy **porównaniu strukturalnym** **niejawnie** wywoływana jest metoda `equals`.

Domyślna implementacja klasy `Any` sprawdza, czy inny obiekt jest **dokładnie tą samą instancją** co obiekt sprawdzający.

Domyślnie każdy obiekt jest unikalny, dlatego operator `==` w tym przypadku zwraca **false**.

Jeżeli chcemy reprezentować równość obiektów przez ich zawartość, alternatywą są **data class**.

Przy zastosowaniu klas danych porównywane są **pola zawarte w konstruktorze głównym**.

```
1 class Person(val name: String)
2 val p1 = Person("Anna")
3 val p2 = Person("Anna")
4
5 println(p1 == p2)
6 println(p1 === p2)
✓ [18] 99ms
```

false

false

```
1 data class Name(val name: String)
2 val name1 = Name("Rafał")
3 val name2 = Name("Rafał")
4
5 println(name1 == name2)
6 println(name1 === name2)
✓ [19] 249ms
```

true

false

```
1 data class Employee(  
2     val name: String,  
3     private val id: Int  
4 ) {  
5     var contractStatus: String = "FAIL"  
6 }  
7  
8 var em1 = Employee("Rafał", 0)  
9 var em2 = Employee("Rafał", 0)  
10  
11 em2.contractStatus = "OK"  
12  
13 println(em1 == em2)  
14 println(em1 === em2)  
[100]
```

true
false

Główne cechy klas danych to możliwość szybkiego definiowania klas do przechowywania informacji oraz implementacja metod takich jak:

- **toString()** - Zwraca reprezentację tekstową obiektu z wartościami jego właściwości.
- **equals()** - Sprawdza równość dwóch obiektów na podstawie wartości ich właściwości (porównanie strukturalne)
- **hashCode()** - Generuje unikalny hash oparty na wartościach właściwości.
- **copy()** - Tworzy kopię obiektu z możliwością modyfikacji wybranych właściwości.
- **componentN()** - Umożliwiają destrukuryzację obiektów w klasach danych. Każda właściwość głównego konstruktora otrzymuje numerowaną metodę **componentN()**.

Główne cechy klas danych to możliwość szybkiego definiowania klas do przechowywania informacji oraz implementacja metod takich jak:

- **toString()** - Zwraca reprezentację tekstową obiektu z wartościami jego właściwości.
- **equals()** - Sprawdza równość dwóch obiektów na podstawie wartości ich właściwości (porównanie strukturalne)
- **hashCode()** - Generuje unikalny hash oparty na wartościach właściwości.
- **copy()** - Tworzy kopię obiektu z możliwością modyfikacji wybranych właściwości.
- **componentN()** - Umożliwiają destrukuryzację obiektów w klasach danych. Każda właściwość głównego konstruktora otrzymuje numerowaną metodę **componentN()**.

Aby klasa mogła być oznaczona jako **data class**, musi spełniać kilka zasad:

- Konstruktor główny musi zawierać co najmniej jedną właściwość (oznaczoną jako **val** lub **var**).
- Klasa nie może być oznaczona jako **abstract**, **sealed** lub **inner**.
- Można dodać inne metody i właściwości do klasy, ale właściwości niebędące w konstruktorze głównym nie są uwzględniane w generowanych metodach.

Porównywanie obiektów

Ponieważ obie zmienne **a** i **b** przechowują tę samą wartość **1**, wynik porównania **==** to **true**. W przypadku **typów prymitywnych**, takich jak **Int**, Kotlin **automatycznie internuje** (ang. **Cache**) wartości często występujące, w celu poprawy wydajności i oszczędzenia pamięci.

```
1  val a = 1
2  val b = 1
3
4  println(a == b)
5  println(a === b)
✓ [15] 88ms
```

true

true

Porównywanie obiektów

Jeżeli wykorzystamy zmienne **nullable**, dla małych wartości **==** dalej zwróci **true**.

```
1 val a: Int = 127
2 val boxedA: Int? = a
3 val differentBoxedA: Int? = a
4 println(boxedA == differentBoxedA)
5 println(boxedA === differentBoxedA)
[94]
```

true

true

```
1 val a: Int = 128
2 val boxedA: Int? = a
3 val differentBoxedA: Int? = a
4 println(boxedA == differentBoxedA)
5 println(boxedA === differentBoxedA)
[95]
```

true

false


```
4 println(a == b)
5 println(a === b)
```

✓ [15] 88ms



true


true

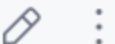
Identity equality for arguments of types Int and Int is deprecated

```
@InlineOnly
public inline fun println(
    message: Boolean
): Unit
```

Prints the given `message` and the line separator to the standard output stream.

 kotlin.io
 ConsoleKt.class

 Mod1



Dziedziczenie

Kotlin wspiera dziedziczenie, ale wymaga, aby **klasa bazowa była jawnie otwarta** na dziedziczenie przez zastosowanie modyfikatora **open**. **Domyślnie** klasy, metody i właściwości w Kotlinie są **final**, co oznacza, że nie mogą być dziedziczone ani nadpisywane.

dostępne do
nadpisanie

```
1 open class Animal(  
2     val name: String,  
3     val body: Int,  
4     val size: Int,  
5     val weight: Int)  
6 {  
7  
8     open fun eat() {  
9         println("Animal.eat() called")  
10    }  
11  
12     open fun move(speed: Int) {  
13         println("Animal.move() called. " +  
14             "Animal is moving at $speed")  
15    }  
16 }
```

Konstruktor klasy pochodnej **musi jawnie wywoływać konstruktor** klasy bazowej.

**jawne wywołanie
konstruktora klasy bazowej**

```
1 ✓ open class Animal(  
2     val name: String,  
3     val body: Int,  
4     val size: Int,  
5     val weight: Int)
```

```
1 ✓ open class Dog(  
2     name: String,  
3     size: Int,  
4     weight: Int,  
5     private val eyes: Int,  
6     private val legs: Int,  
7     private val tail: Int  
8 ) : Animal(name, 1, size, weight) {  
9  
10 ✓ override fun eat() {  
11     println("Dog.eat() called")  
12     super.eat()  
13 }  
14  
15 ✓ final override fun move(speed: Int) {  
16     println("Dog.move() called")  
17     super.move(speed)  
18 }  
19 }
```

Konstruktor klasy pochodnej **musi jawnie wywoływać konstruktor** klasy bazowej.

**jawne wywołanie
konstruktora klasy bazowej**

```
1 open class Animal(  
2     val name: String,  
3     val body: Int,  
4     val size: Int,  
5     val weight: Int)
```

**element nadpisujący jest
domyślnie otwarty**

```
open fun eat() {  
    println("Animal.eat() called")  
}
```

```
1 open class Dog(  
2     name: String,  
3     size: Int,  
4     weight: Int,  
5     private val eyes: Int,  
6     private val legs: Int,  
7     private val tail: Int  
8 ) : Animal(name, 1, size, weight) {  
9  
10    override fun eat() {  
11        println("Dog.eat() called")  
12        super.eat()  
13    }  
14  
15    final override fun move(speed: Int) {  
16        println("Dog.move() called")  
17        super.move(speed)  
18    }  
19 }
```

Konstruktor klasy pochodnej **musi jawnie wywoływać konstruktor** klasy bazowej.

**jawne wywołanie
konstruktora klasy bazowej**

```
1 open class Animal(  
2     val name: String,  
3     val body: Int,  
4     val size: Int,  
5     val weight: Int)
```

**element nadpisujący jest
domyślnie otwarty**

```
open fun eat() {  
    println("Animal.eat() called")  
}
```

**wywołanie metody klasy
bazowej**

```
1 open class Dog(  
2     name: String,  
3     size: Int,  
4     weight: Int,  
5     private val eyes: Int,  
6     private val legs: Int,  
7     private val tail: Int  
8 ) : Animal(name, 1, size, weight) {  
9  
10    override fun eat() {  
11        println("Dog.eat() called")  
12        super.eat()  
13    }  
14  
15    final override fun move(speed: Int) {  
16        println("Dog.move() called")  
17        super.move(speed)  
18    }  
19 }
```

Konstruktor klasy pochodnej **musi jawnie wywoływać konstruktor** klasy bazowej.

**jawne wywołanie
konstruktora klasy bazowej**

```
1 open class Animal(  
2     val name: String,  
3     val body: Int,  
4     val size: Int,  
5     val weight: Int)
```

**element nadpisujący jest
domyślnie otwarty**

```
open fun eat() {  
    println("Animal.eat() called")  
}
```

**wywołanie metody klasy
bazowej**

**uniemożliwienie dalsze
nadpisywanie w kolejnych
klasach pochodnych.**

```
open fun move(speed: Int) {  
    println("Animal.move() called. " +  
        "Animal is moving at $speed")  
}
```

```
1 open class Dog(  
2     name: String,  
3     size: Int,  
4     weight: Int,  
5     private val eyes: Int,  
6     private val legs: Int,  
7     private val tail: Int  
8 ) : Animal(name, 1, size, weight) {  
9  
10    override fun eat() {  
11        println("Dog.eat() called")  
12        super.eat()  
13    }  
14  
15    final override fun move(speed: Int) {  
16        println("Dog.move() called")  
17        super.move(speed)  
18    }  
19 }
```

Klasy abstrakcyjne

Klasy abstrakcyjne pozwalają na definiowanie **wspólnego interfejsu i zachowania** dla grupy klas. Słowo kluczowe **abstract** oznacza, że nie można bezpośrednio utworzyć instancji klasy. Służy wyłącznie jako **szablon dla klas pochodnych**.

Metody abstrakcyjne nie mają implementacji. Klasy pochodne są zobowiązane do ich nadpisania.

```
1 abstract class Animal {  
2  
3     abstract fun eat()  
4  
5     open fun move(){  
6         println("Animal.move() called")  
7     }  
8 }  
9  
10 class Dog (): Animal(){  
11     final override fun eat(){  
12         println("Dog.eat() called")  
13     }  
14 }
```

Klasy abstrakcyjne

Klasy abstrakcyjne pozwalają na definiowanie **wspólnego interfejsu i zachowania** dla grupy klas. Słowo kluczowe **abstract** oznacza, że nie można bezpośrednio utworzyć instancji klasy. Służy wyłącznie jako **szablon dla klas pochodnych**.

Metody abstrakcyjne nie mają implementacji. Klasy pochodne są zobowiązane do ich nadpisania.

Może zawierać również metody i właściwości z domyślną implementacją, które klasy pochodne mogą nadpisywać

```
1 abstract class Animal {  
2  
3     abstract fun eat()  
4  
5     open fun move(){  
6         println("Animal.move() called")  
7     }  
8 }  
9  
10 class Dog (): Animal(){  
11     final override fun eat(){  
12         println("Dog.eat() called")  
13     }  
14 }
```


Klasy abstrakcyjne

Klasy abstrakcyjne pozwalają na definiowanie **wspólnego interfejsu i zachowania** dla grupy klas. Słowo kluczowe **abstract** oznacza, że nie można bezpośrednio utworzyć instancji klasy. Służy wyłącznie jako **szablon dla klas pochodnych**.

Metody abstrakcyjne nie mają implementacji. Klasy pochodne są zobowiązane do ich nadpisania.

Może zawierać również metody i właściwości z domyślną implementacją, które klasy pochodne mogą nadpisywać

wymagana implementacja

```
1 abstract class Animal {  
2  
3     abstract fun eat()  
4  
5     open fun move(){  
6         println("Animal.move() called")  
7     }  
8 }  
9  
10 class Dog (): Animal(){  
11     final override fun eat(){  
12         println("Dog.eat() called")  
13     }  
14 }
```

Klasy abstrakcyjne

Klasy abstrakcyjne pozwalają na definiowanie **wspólnego interfejsu i zachowania** dla grupy klas. Słowo kluczowe **abstract** oznacza, że nie można bezpośrednio utworzyć instancji klasy. Służy wyłącznie jako **szablon dla klas pochodnych**.

Metody abstrakcyjne nie mają implementacji. Klasy pochodne są zobowiązane do ich nadpisania.

Może zawierać również metody i właściwości z domyślną implementacją, które klasy pochodne mogą nadpisywać

wymagana implementacja

- **final** - domyślny modyfikator, nie można nadpisać
- **open** - można nadpisać
- **abstract** - wymagane nadpisanie
- **override** - nadpisuje element klasy nadrzędnej

```
1  abstract class Animal {  
2  
3      abstract fun eat()  
4  
5      open fun move(){  
6          println("Animal.move() called")  
7      }  
8  }  
9  
10 class Dog (): Animal(){  
11     final override fun eat(){  
12         println("Dog.eat() called")  
13     }  
14 }
```

Klasy zagnieżdżone i wewnętrzne

W Kotlinie klasy mogą być **zagnieżdżane** wewnątrz innych klas. Dostępne są dwa rodzaje takich klas: **zagnieżdżone** i **wewnętrzne**, które różnią się dostępem do elementów klasy zewnętrznej.

```
1 class Outer {  
2     private val outerValue = "Value from Outer"  
3  
4     // Klasa zagnieżdżona  
5     class Nested {  
6         fun nestedFunction() = "Called from Nested"  
7     }  
8 }  
9  
10 // Tworzenie instancji klasy zagnieżdżonej  
11 val nested = Outer.Nested()
```

Klasa **zagnieżdżona** to klasa, która jest zadeklarowana wewnątrz innej klasy. Domyślnie klasa zagnieżdżona **nie ma dostępu** do elementów klasy zewnętrznej. Jest traktowana jak **niezależny byt**.

Klasy zagnieżdżone i wewnętrzne

W Kotlinie klasy mogą być **zagnieżdżane** wewnątrz innych klas. Dostępne są dwa rodzaje takich klas: **zagnieżdżone** i **wewnętrzne**, które różnią się dostępem do elementów klasy zewnętrznej.

Klasa **zagnieżdżona** to klasa, która jest zadeklarowana wewnątrz innej klasy. Domyślnie klasa zagnieżdżona **nie ma dostępu** do elementów klasy zewnętrznej. Jest traktowana jak **niezależny byt**.

```
1 class Outer {  
2     private val outerValue = "Value from Outer"  
3  
4     // Klasa zagnieżdżona  
5     class Nested {  
6         fun nestedFunction() = "Called from Nested"  
7     }  
8 }  
9
```

```
10 // Tworzenie instancji klasy zag  
11 val nested = Outer.Nested()
```

Klasa **wewnętrzna** to klasa, która jest zadeklarowana wewnątrz innej klasy i **ma dostęp do jej pól oraz metod**. Należy oznaczyć ją słowem kluczowym **inner**.

```
1 class Outer {  
2     private val outerValue = "Value from Outer"  
3  
4     // Klasa wewnętrzna  
5     inner class Inner {  
6         fun innerFunction() = "Accessing: $outerValue"  
7     }  
8 }  
9
```

```
10 // Tworzenie instancji klasy wewnętrznej  
11 val inner = Outer().Inner()
```

Klasy zapieczętowane

Kotlin wprowadza mechanizm **klas zapieczętowanych (sealed classes)**, który jest przydatny w sytuacjach, gdy chcemy **ograniczyć hierarchię dziedziczenia** do z góry określonego zbioru klas.

klasa zapieczętowana

podklasy mogą być
zdefiniowane tylko wewnątrz
tego samego pliku, co klasa
bazowa

```
sealed class Response
```

```
2 class Success(val data: String) : Response()
```

```
3 class Error(val error: String) : Response()
```

```
4 class Loading : Response()
```

```
1 fun handleResponse(response: Response) {  
2     when (response) {  
3         is Success -> println("Data: ${response.data}")  
4         is Error -> println("Error: ${response.error}")  
5         is Loading -> println("Loading...")  
6     }  
7 }
```

Klasy zapieczętowane

Kotlin wprowadza mechanizm **klas zapieczętowanych (sealed classes)**, który jest przydatny w sytuacjach, gdy chcemy **ograniczyć hierarchię dziedziczenia** do z góry określonego zbioru klas.

klasa zapieczętowana

podklasy mogą być zdefiniowane tylko wewnątrz tego samego pliku, co klasa bazowa

```
sealed class Response
```

```
2 class Success(val data: String) : Response()
```

```
3 class Error(val error: String) : Response()
```

```
4 class Loading : Response()
```

Mamy trzy możliwe stany:

- **Success** – oznacza powodzenie i zawiera dane
- **Error** – oznacza błąd i zawiera komunikat o błędzie
- **Loading** – oznacza trwające ładowanie

```
1 fun handleResponse(response: Response) {  
2     when (response) {  
3         is Success -> println("Data: ${response.data}")  
4         is Error -> println("Error: ${response.error}")  
5         is Loading -> println("Loading...")  
6     }  
7 }
```

Klasy zapieczętowane

Kotlin wprowadza mechanizm **klas zapieczętowanych (sealed classes)**, który jest przydatny w sytuacjach, gdy chcemy **ograniczyć hierarchię dziedziczenia** do z góry określonego zbioru klas.

klasa zapieczętowana

podklasy mogą być zdefiniowane tylko wewnątrz tego samego pliku, co klasa bazowa

```
sealed class Response
```

```
2 class Success(val data: String) : Response()
```

```
3 class Error(val error: String) : Response()
```

```
4 class Loading : Response()
```

Mamy trzy możliwe stany:

- **Success** – oznacza powodzenie i zawiera dane
- **Error** – oznacza błąd i zawiera komunikat o błędzie
- **Loading** – oznacza trwające ładowanie

Dzięki klasom zapieczętowanym możemy bezpiecznie obsługiwać **wszystkie możliwe stany**

wszystkie możliwe stany

brak bloku **else**

```
1 fun handleResponse(response: Response) {  
2     when (response) {  
3         is Success -> println("Data: ${response.data}")  
4         is Error -> println("Error: ${response.error}")  
5         is Loading -> println("Loading...")  
6     }  
7 }
```


enum class to specjalny typ klasy, który służy do **definiowania zbioru stałych**.

```
1  enum class Direction {  
2      NORTH, SOUTH, EAST, WEST  
3  }
```

```
1  fun navigate(direction: Direction) {  
2      when (direction) {  
3          Direction.NORTH -> println("Idziesz na północ")  
4          Direction.SOUTH -> println("Idziesz na południe")  
5          Direction.EAST -> println("Idziesz na wschód")  
6          Direction.WEST -> println("Idziesz na zachód")  
7      }  
8  }  
9  
10 val currentDirection = Direction.NORTH  
11 navigate(currentDirection) // Output: Idziesz na północ
```


Klasy generyczne

słowo
kluczowe

nazwa
klasy

parametr
typowany

nazwa
parametru

typ
parametru

```
1 class Box<T>(val value: T) {  
2     fun getVal(): T = value  
3 }  
4
```

```
5 val intBox = Box(42)  
6 val stringBox = Box("Hello")  
7 println(intBox.getVal())  
8 println(stringBox.getVal())
```

```
✓ [38] 133ms
```

42

Hello



Klasy generyczne

Kotlin wprowadza **wariancję** dla **klas generycznych**, która określa, czy typy generyczne są **kowariantne** (zachowują zgodność typów w jednym kierunku) czy **kontrawariantne** (działają w przeciwnym kierunku).

out oznacza, że generyczny typ może być jedynie **produkowany** (czyli używany jako **typ zwracany**)

```
1 class Producer<out T>(private val value: T) {  
2     fun get(): T = value  
3 }
```

typ zwracany

określony typ



Producer<out T>



dowolny typ

Klasy generyczne

Kotlin wprowadza **wariancję** dla **klas generycznych**, która określa, czy typy generyczne są **kowariantne** (zachowują zgodność typów w jednym kierunku) czy **kontrawariantne** (działają w przeciwnym kierunku).

out oznacza, że generyczny typ może być jedynie **produkowany** (czyli używany jako **typ zwracany**)

```
1 class Producer<out T>(private val value: T) {  
2     fun get(): T = value  
3 }
```

typ zwracany

in oznacza, że generyczny typ może być jedynie **konsumowany** (używany jako **typ argumentu**)

```
1 class Consumer<in T> {  
2     fun consume(value: T) {  
3         println("Consumed: $value")  
4     }  
5 }
```

typ przyjmowany

określony typ

Producer<out T>

dowolny typ

Consumer<in T>

określony typ