



PROGRAMOWANIE URZĄDZEŃ MOBILNYCH 1

WYKŁAD 6

- Obiekty i Interfejsy

Słowo kluczowe **object** pozwala na jednoczesne zdefiniowanie klasy i utworzenie jej instancji.

Jest to przydatne w dwóch głównych scenariuszach: tworzeniu **singletonów** oraz **anonimowych obiektów**.

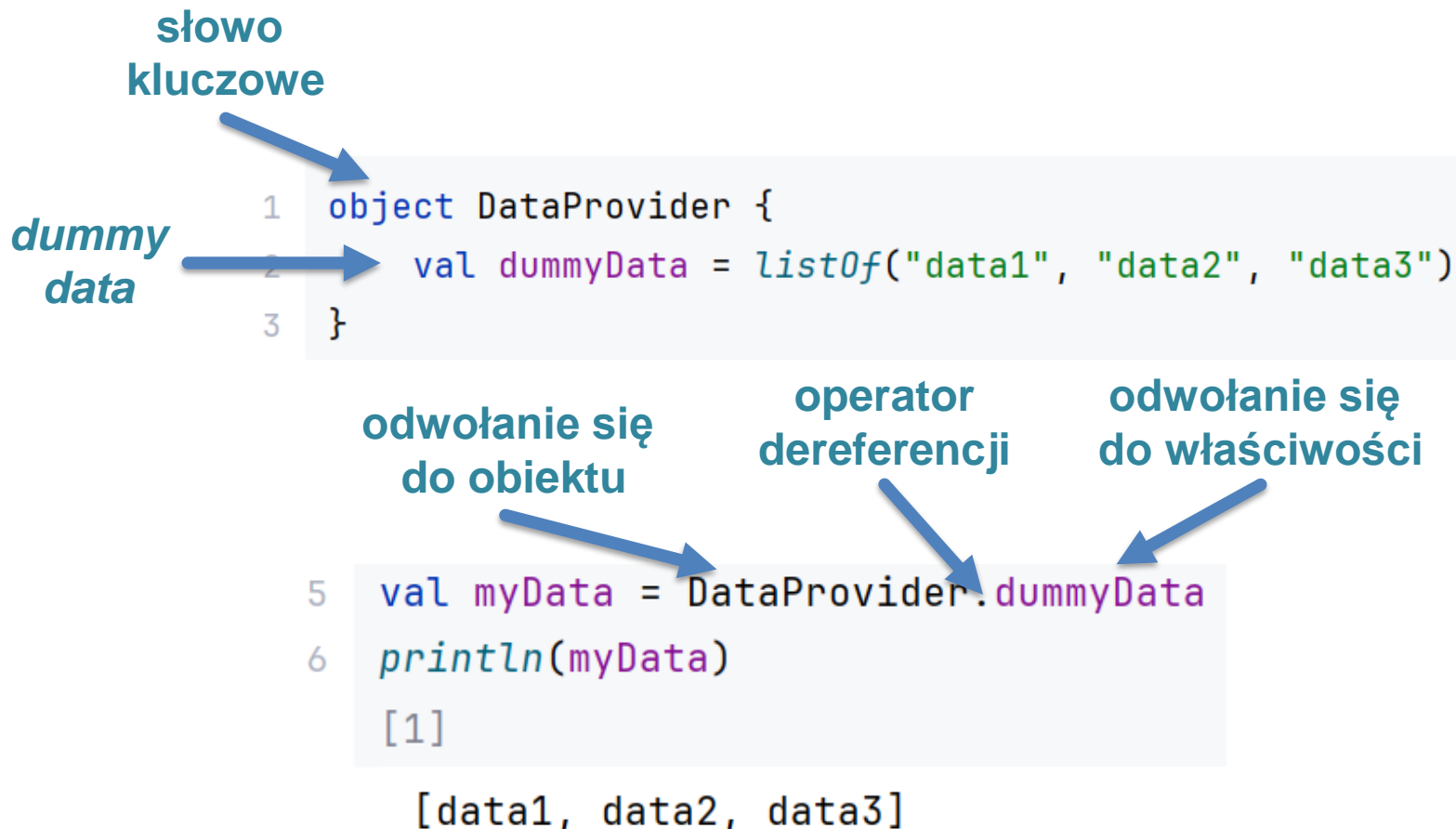
słowo
kluczowe

*dummy
data*

```
1 object DataProvider {  
2   val dummyData = listOf("data1", "data2", "data3")  
3 }
```

Słowo kluczowe **object** pozwala na jednoczesne zdefiniowanie klasy i utworzenie jej instancji.

Jest to przydatne w dwóch głównych scenariuszach: tworzeniu **singletonów** oraz **anonimowych obiektów**.



Możemy uzyskiwać dostęp do właściwości `dummyData` bez potrzeby jawnego tworzenia instancji klasy.

Obiekty anonimowe można wykorzystać do **jednorazowej** implementacji interfejsów lub klas abstrakcyjnych.

wymagana
implementacja

```
1 abstract class Shape {  
2     abstract fun draw()  
3 }
```

Obiekty anonimowe można wykorzystać do **jednorazowej** implementacji interfejsów lub klas abstrakcyjnych.

wymagana
implementacja

```
1 abstract class Shape {  
2     abstract fun draw()  
3 }
```

Przypisanie instancji
obektu anonimowego

```
5 val circle = object : Shape() {  
6     override fun draw() {  
7         println("Drawing a circle")  
8     }  
9 }
```

Obiekty anonimowe można wykorzystać do **jednorazowej** implementacji interfejsów lub klas abstrakcyjnych.

wymagana
implementacja

```
1 abstract class Shape {  
2     abstract fun draw()  
3 }
```

Utworzenie i zainicjowanie **obiektu anonimowego**

Przypisanie **instancji obiektu anonimowego**

```
5 val circle = object : Shape() {  
6     override fun draw() {  
7         println("Drawing a circle")  
8     }  
9 }
```

Obiekty anonimowe można wykorzystać do **jednorazowej** implementacji interfejsów lub klas abstrakcyjnych.

wymagana
implementacja

```
1 abstract class Shape {  
2     abstract fun draw()  
3 }
```

Utworzenie i zainicjowanie **obektu anonimowego**

Przypisanie **instancji objektu anonimowego**

Obiekt **dziedziczy** po klasie **Shape**

```
5 val circle = object : Shape() {  
6     override fun draw() {  
7         println("Drawing a circle")  
8     }  
9 }
```

Obiekty anonimowe można wykorzystać do **jednorazowej** implementacji interfejsów lub klas abstrakcyjnych.

wymagana
implementacja

```
1 abstract class Shape {  
2     abstract fun draw()  
3 }
```

Utworzenie i zainicjowanie **obektu anonimowego**

Przypisanie **instancji obektu anonimowego**

Obiekt **dziedziczy** po klasie **Shape**

```
5 val circle = object : Shape() {  
6     override fun draw() {  
7         println("Drawing a circle")  
8     }  
9 }
```

Implementacja metody abstrakcyjnej

Obiekty danych rozszerza funkcjonalność zwykłych obiektów, przez wprowadzenie

W obiektach danych automatyczne generowanie metod **equals()** i **hashCode()**. Wspiera porównania strukturalne, co pozwala porównywać obiekty tego samego typu jako równe, oraz wykorzystanie w kolekcjach opartych o *hashowanie*.

Różna implementacja metody **toString()**

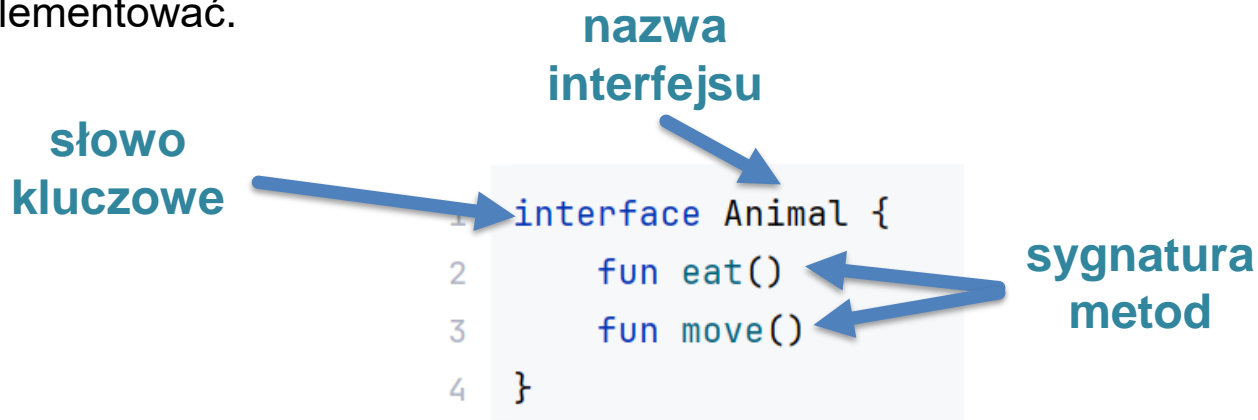
```
1 ✓ data object MyDataObject {  
2     val number: Int = 3  
3 }  
4  
5 ✓ object MyObject {  
6     val number: Int = 3  
7 }  
8  
9 println(MyObject)  
10 println(MyDataObject)  
[3]
```

Line_9_jupyter\$MyObject@7562a580
MyDataObject

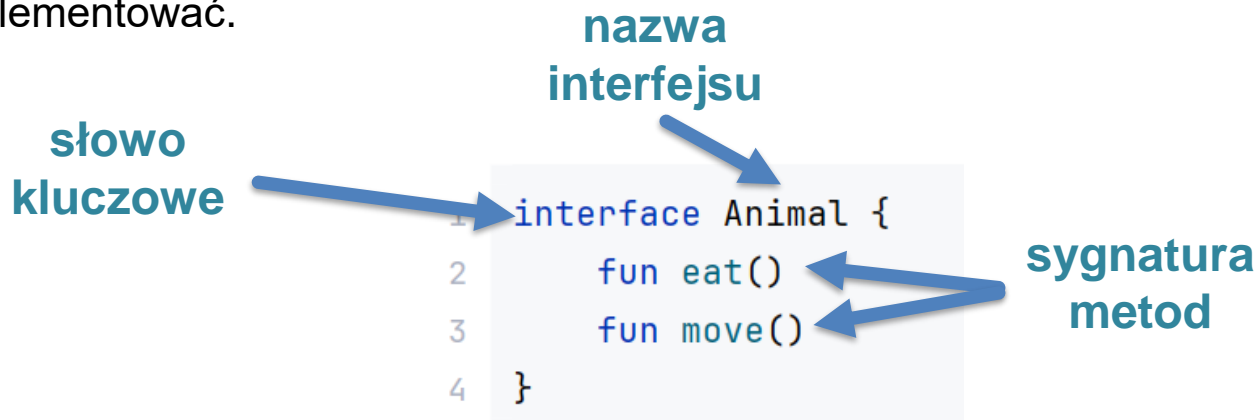
Interfejs to struktura, która definiuje zestaw metod i właściwości, które klasy mogą implementować.

```
1 interface Animal {  
2     fun eat()  
3     fun move()  
4 }
```

Interfejs to struktura, która definiuje zestaw metod i właściwości, które klasy mogą implementować.

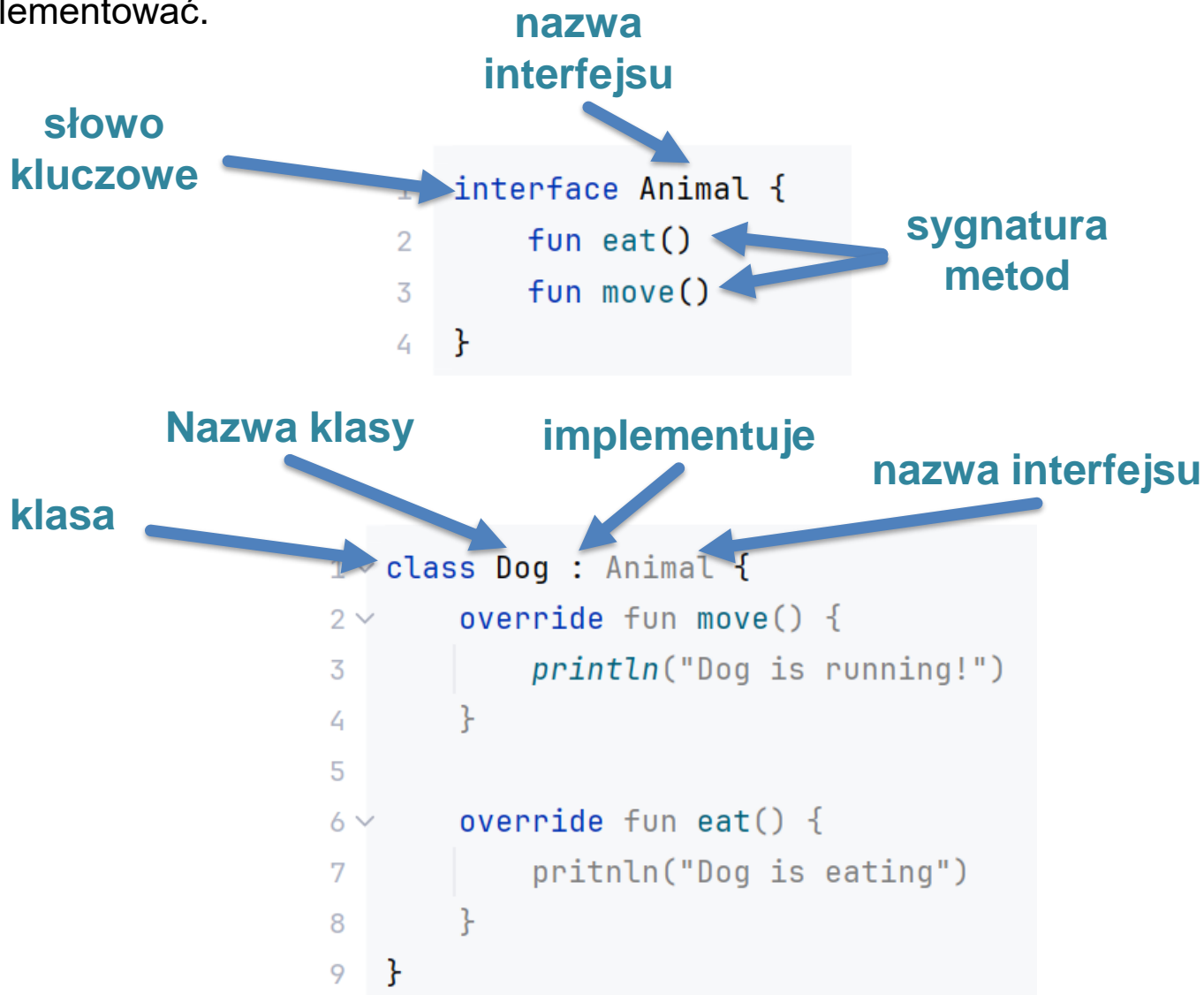


Interfejs to struktura, która definiuje zestaw metod i właściwości, które klasy mogą implementować.

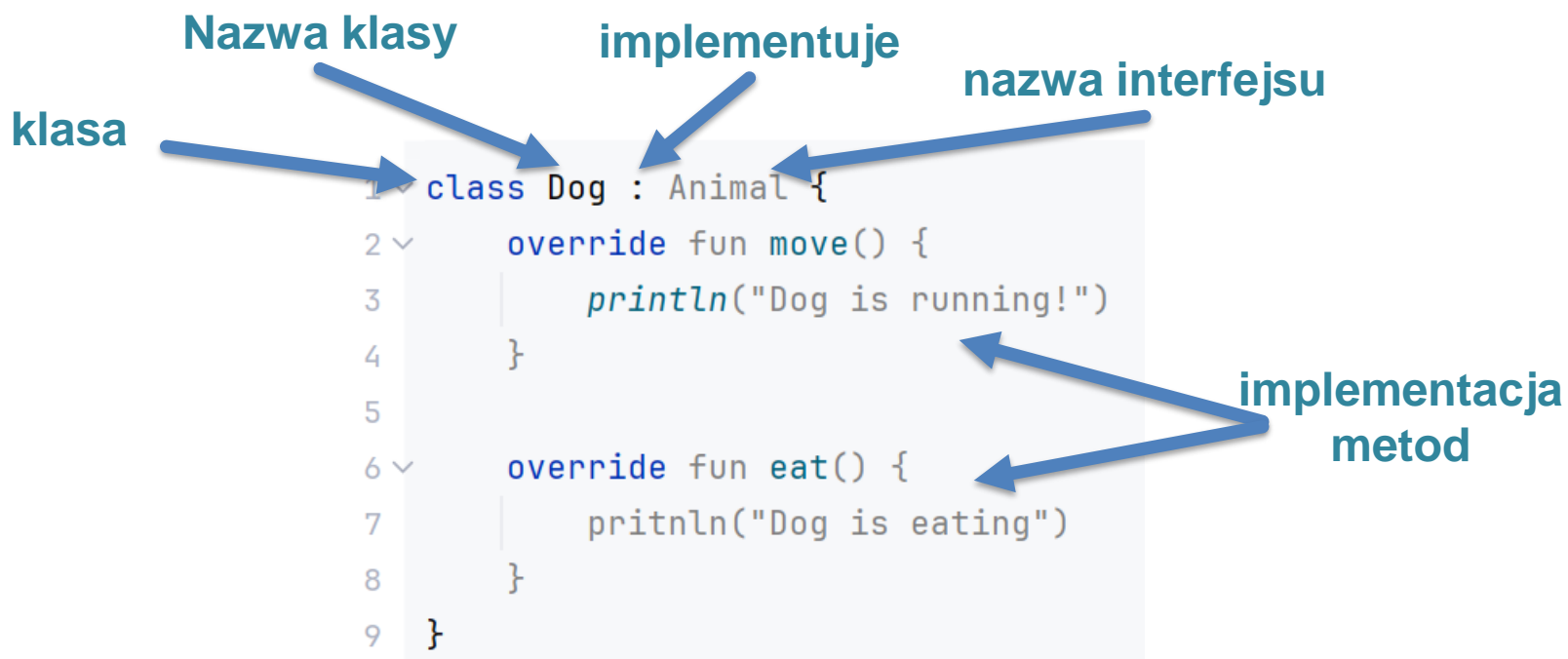
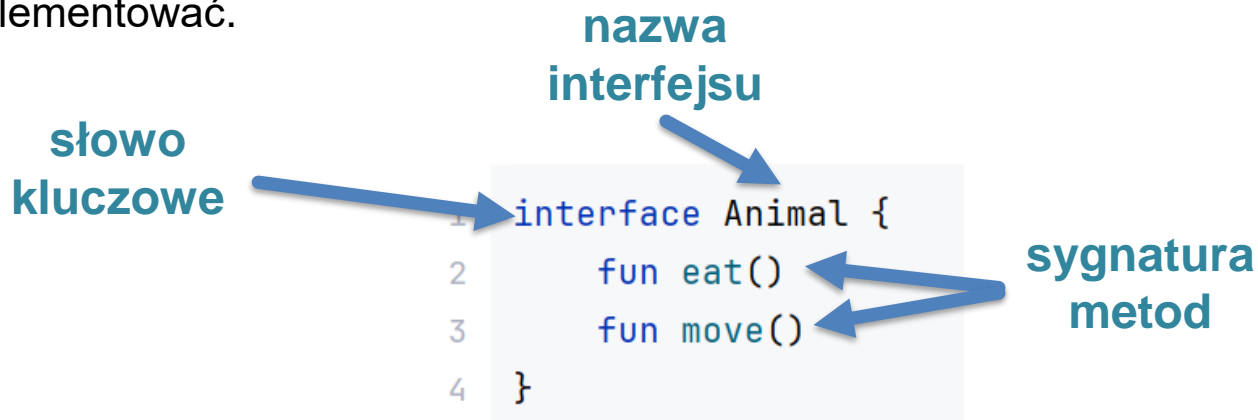


```
1 ✓ class Dog : Animal {  
2 ✓     override fun move() {  
3         println("Dog is running!")  
4     }  
5  
6 ✓     override fun eat() {  
7         println("Dog is eating")  
8     }  
9 }
```

Interfejs to struktura, która definiuje zestaw metod i właściwości, które klasy mogą implementować.



Interfejs to struktura, która definiuje zestaw metod i właściwości, które klasy mogą implementować.



Klasy mogą **dziedziczyć** tylko **po jednej klasie**, ale mogą **implementować dowolną ilość interfejsów**.

```
1 interface Student { fun getIndexNumber(): Int }  
2 interface Human { fun getName(): String }
```

Klasy mogą **dziedziczyć** tylko **po jednej klasie**, ale mogą **implementować dowolną ilość interfejsów**.

```
1 interface Student { fun getIndexNumber(): Int }
2 interface Human { fun getName(): String }
```

Nazwa klasy
klasa →

implementowane interfejsy

```
4 class ISSPStudent : Student, Human {
5
6     override fun getIndexNumber() = 123456
7     override fun getName() = "Rafał"
8 }
```

implementacja metod →

Interfejsy mogą dostarczać **domyślną** implementację metod.

Metoda z **domyślną** implementacją.

Nie wymagane nadpisanie przez klasy implementujące

Wymagane nadpisanie przez klasy implementujące

```
1 interface Animal {  
2     fun eat() {  
3         println("Eating...")  
4     }  
5  
6     fun move()  
7 }
```

Interfejsy mogą dostarczać **domyślną** implementację metod.

Metoda z **domyślną** implementacją.

Nie wymagane nadpisanie przez klasy implementujące

Wymagane nadpisanie przez klasy implementujące

```
1 interface Animal {  
2     fun eat() {  
3         println("Eating...")  
4     }  
5  
6     fun move()  
7 }
```

Klasa **implementuje** interfejs

Wymagana implementacja metody

```
9 class Dog : Animal {  
10     override fun move() {  
11         println("Dog is running!")  
12     }  
13 }
```

Kotlin pozwala również na definiowanie **metod prywatnych** w interfejsach.

Metoda prywatna jest dostępna wyłącznie w **obrębie interfejsu** i mogą być używane do wspierania implementacji metod publicznych.

```
1 interface Logger {  
2     fun logInfo(message: String) {  
3         log("INFO: $message")  
4     }  
5  
6     private fun log(message: String) {  
7         println(message)  
8     }  
9 }
```

Kotlin pozwala również na definiowanie **metod prywatnych** w interfejsach.

Metoda prywatna jest dostępna wyłącznie w **obrębie interfejsu** i mogą być używane do wspierania implementacji metod publicznych.

```
1 interface Logger {  
2     fun logInfo(message: String) {  
3         log("INFO: $message")  
4     }  
5  
6     private fun log(message: String) {  
7         println(message)  
8     }  
9 }
```

Mogą posiadać **właściwości**.

Właściwość ze zdefiniowanym
getterem

```
1 interface Numbers{  
2     val num: Int get() = 4  
3 }  
4  
5 val number = object : Numbers{  
6     println(number.num)
```

Dziedziczenie interfejsów

Interfejsy **mogą dziedziczyć** po innych interfejsach. Różnicą jest możliwość **wielokrotnego dziedziczenia**.

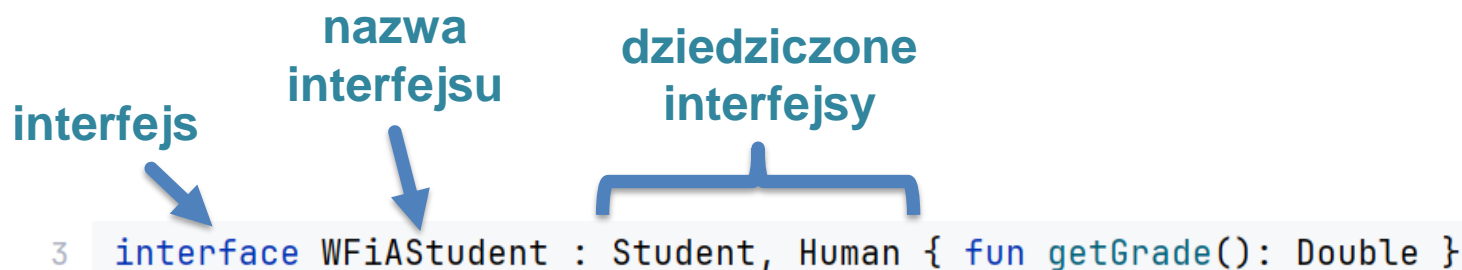
```
1 interface Student { fun getIndexNumber(): Int }  
2 interface Human { fun getName(): String }
```

Dziedziczenie interfejsów

Interfejsy **mogą dziedziczyć** po innych interfejsach. Różnicą jest możliwość **wielokrotnego dziedziczenia**.

```
1 interface Student { fun getIndexNumber(): Int }  
2 interface Human { fun getName(): String }
```

interfejs nazwa interfejsu dziedziczone interfejsy



```
3 interface WFiAStudent : Student, Human { fun getGrade(): Double }
```

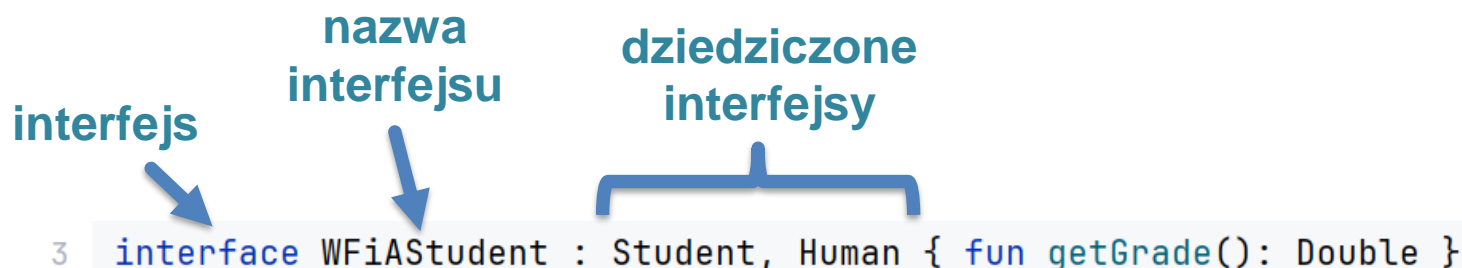
Dziedziczenie interfejsów

Interfejsy **mogą dziedziczyć** po innych interfejsach. Różnicą jest możliwość **wielokrotnego dziedziczenia**.

```
1 interface Student { fun getIndexNumber(): Int }
2 interface Human { fun getName(): String }
```

interfejs nazwa interfejsu dziedziczone interfejsy

```
3 interface WFiAStudent : Student, Human { fun getGrade(): Double }
```



Klasa implementująca interfejs musi implementować **wszystkie metody** interfejsów dziedziczonych.

```
5 class ISSPStudent : WFiAStudent{
6
7     override fun getIndexNumber() = 123456
8     override fun getName() = "Rafał"
9     override fun getGrade() = 4.5
10 }
```

Ustalenie sygnatury metod

Jednym z głównych zastosowań interfejsów jest **narzucenie klasom implementującym** interfejs **obowiązku** zaimplementowania metod zadeklarowanych w interfejsie.

```
1 interface Animal {  
2     fun eat()  
3 }  
4  
5 class Dog : Animal {  
6     override fun eat() { println("Dog is eating") }  
7 }  
8  
9 class Bird : Animal {  
10     override fun eat() { println("Bird is eating seeds") }  
11 }
```


Polimorfizm

Możemy używać **interfejsów jako typów**, obiekty różnych klas mogą być **traktowane w jednolity sposób**, o ile implementują ten sam interfejs.

```
1  interface Shape { fun draw() }
2
3  class Circle : Shape {
4      override fun draw() { println("Drawing a Circle") }
5  }
6
7  class Square : Shape {
8      override fun draw() { println("Drawing a Square") }
9  }
```

Polimorfizm

Możemy używać **interfejsów jako typów**, obiekty różnych klas mogą być **traktowane w jednolity sposób**, o ile implementują ten sam interfejs.

```
1 interface Shape { fun draw() }
2
3 class Circle : Shape {
4     override fun draw() { println("Drawing a Circle") }
5 }
6
7 class Square : Shape {
8     override fun draw() { println("Drawing a Square") }
9 }
```

```
11 fun render(shape: Shape) { shape.draw() }
```

Obiekt o **typie** interfejsu



```
13 val shapes: List<Shape> = listOf(Circle(), Square())
```

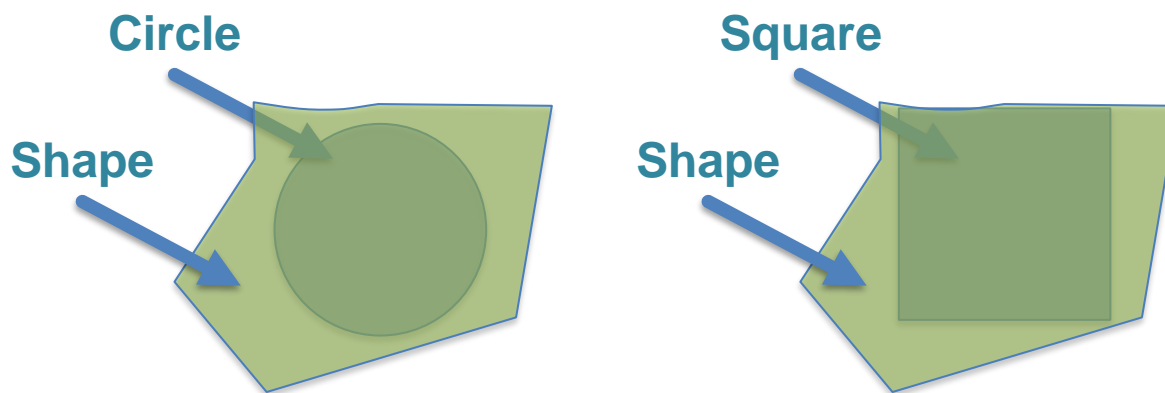
Polimorfizm

Możemy używać **interfejsów jako typów**, obiekty różnych klas mogą być **traktowane w jednolity sposób**, o ile implementują ten sam interfejs.

```
11 fun render(shape: Shape) { shape.draw() }
```

Obiekt o **typie** interfejsu

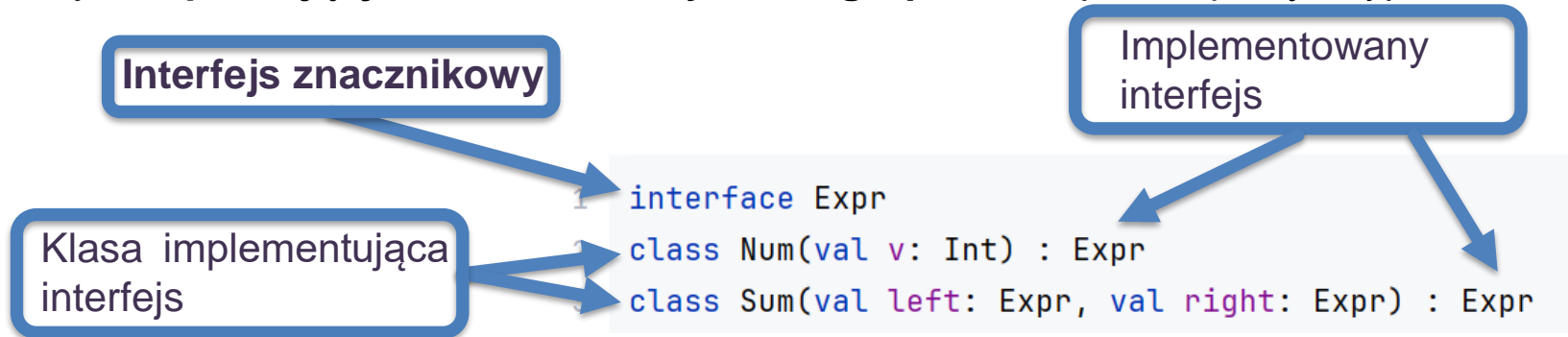
```
13 val shapes: List<Shape> = listOf(Circle(), Square())
```



Implementowanie interfejsu nadaje klasie typ interfejsu

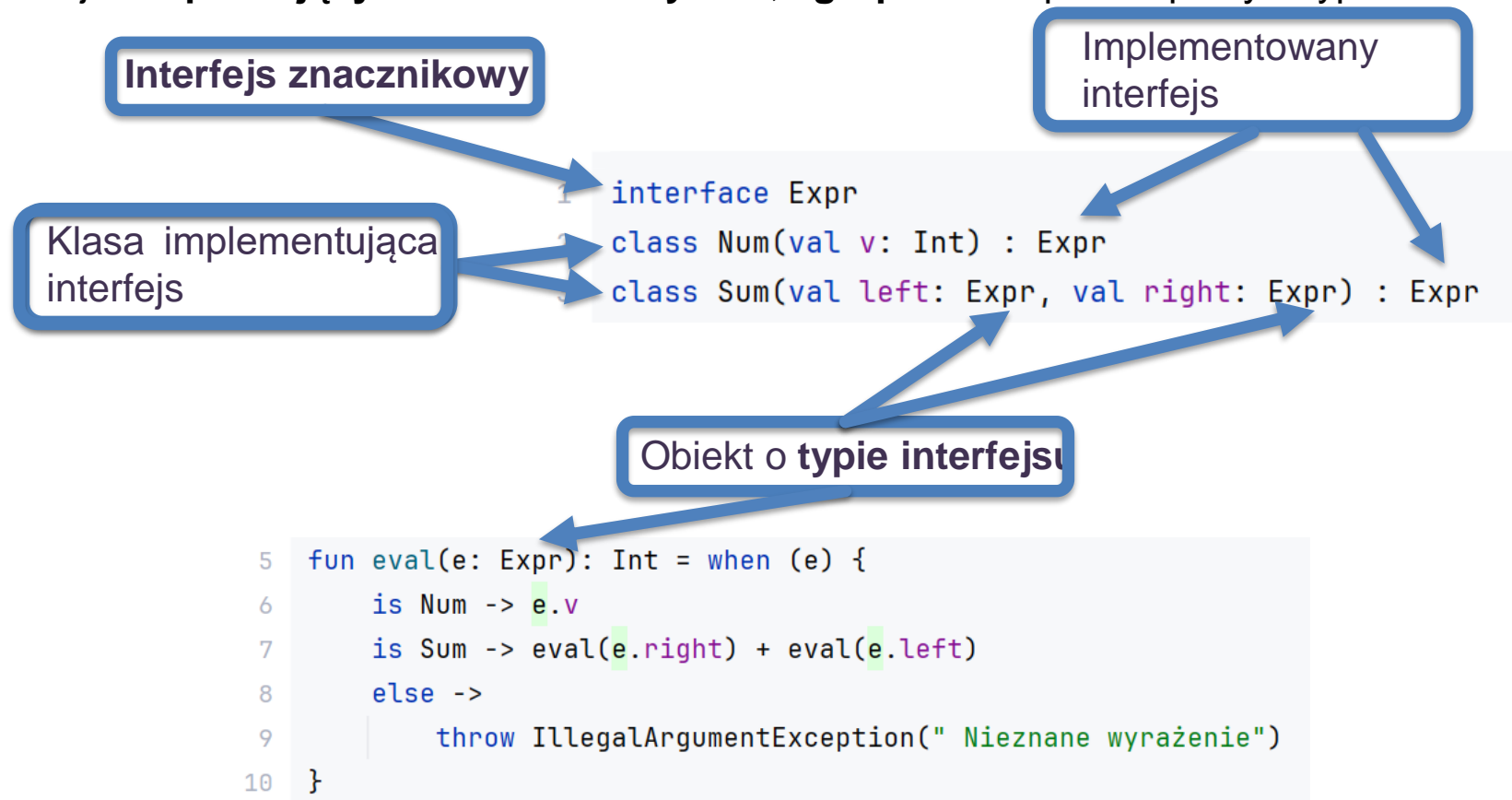
Interfejs znacznikowy

interfejs, który **nie zawiera żadnych** metod ani właściwości. Celem jest **oznaczenie** klas jako **spełniających określone kryteria**, **zgrupowanie** pod wspólnym typem.



Interfejs znacznikowy

interfejs, który **nie zawiera żadnych** metod ani właściwości. Celem jest **oznaczenie** klas jako **spełniających określone kryteria**, **zgrupowanie** pod wspólnym typem.



Przykładem interfejsu znacznikowego w bibliotece standardowej jest **Serializable**.

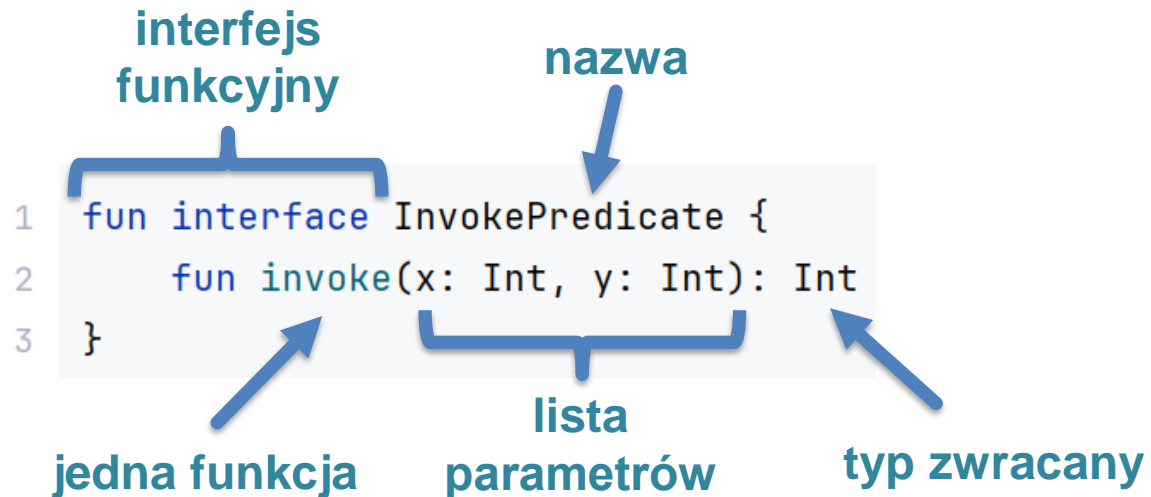
Nadanie wspólnej cechy

interfejsy **opisują cechę** (na podstawie **metod** i **właściwości**), która jest **nadana** implementującym je klasom.

- Comparable
- Serializable
- Iterable
- Closable
- Callable
- Runnable

Interfejs funkcyjny

Zawierają **tylko jedną metodę**, są one wykorzystywane głównie do reprezentowania **wyrażeń lambda** jako **typów danych**.



```
1 fun interface InvokePredicate {  
2     fun invoke(x: Int, y: Int): Int  
3 }
```

The diagram illustrates the components of the `InvokePredicate` functional interface. Annotations with arrows point to specific parts of the code:

- interfejs funkcyjny**: Points to the `fun interface` keyword.
- nazwa**: Points to the interface name `InvokePredicate`.
- jedna funkcja**: Points to the `fun invoke` method signature.
- lista parametrów**: Points to the parameters `(x: Int, y: Int)`.
- typ zwracany**: Points to the return type `: Int`.

Interfejs funkcyjny

Zawierają **tylko jedną metodę**, są one wykorzystywane głównie do reprezentowania **wyrażeń lambda** jako **typów danych**.

interfejs funkcyjny

```
1 fun interface InvokePredicate {  
2     fun invoke(x: Int, y: Int): Int  
3 }
```

nazwa

jedna funkcja

lista parametrów

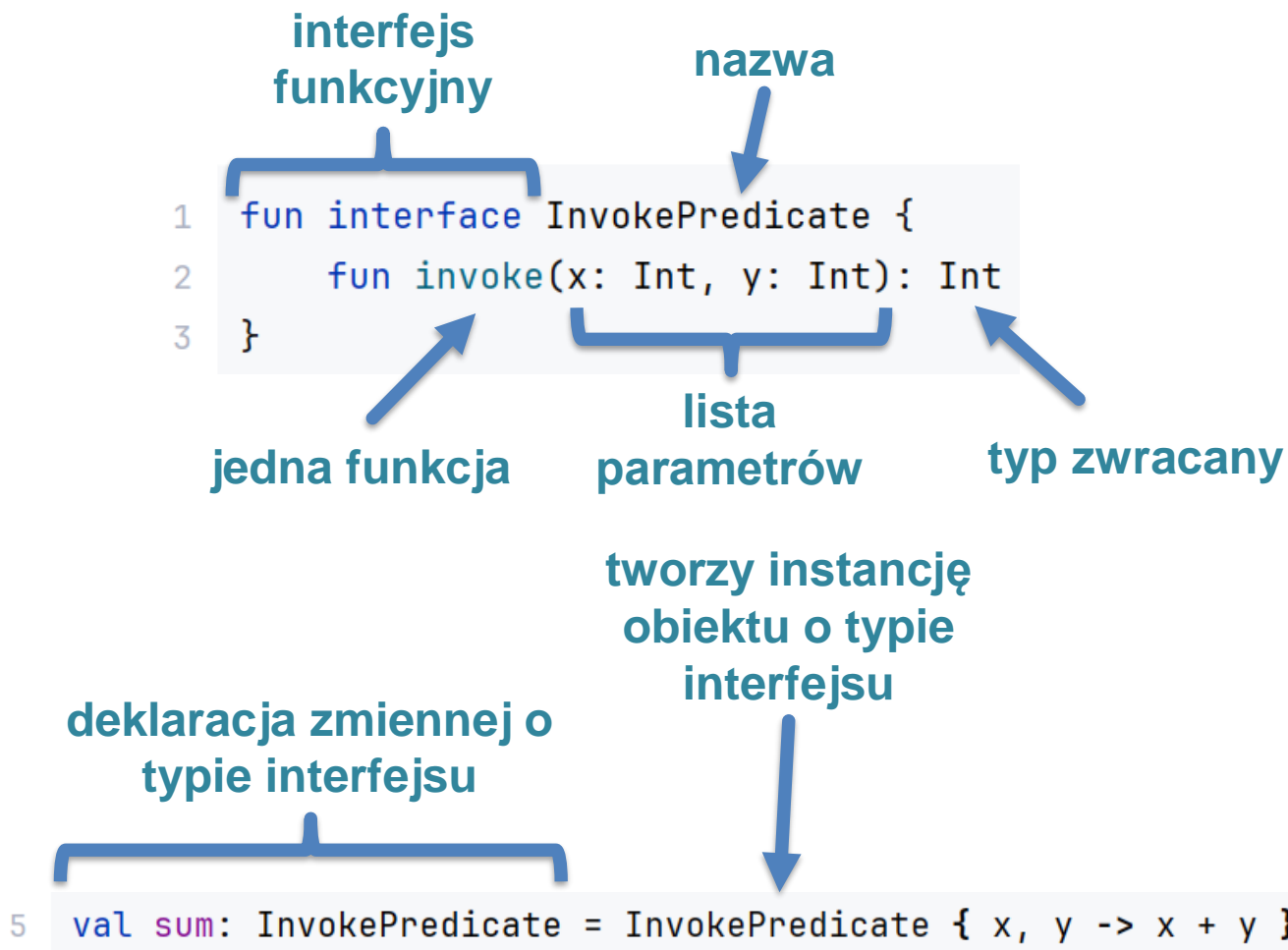
typ zwracany

**deklaracja zmiennej o
typie interfejsu**

```
5 val sum: InvokePredicate = InvokePredicate { x, y -> x + y }
```

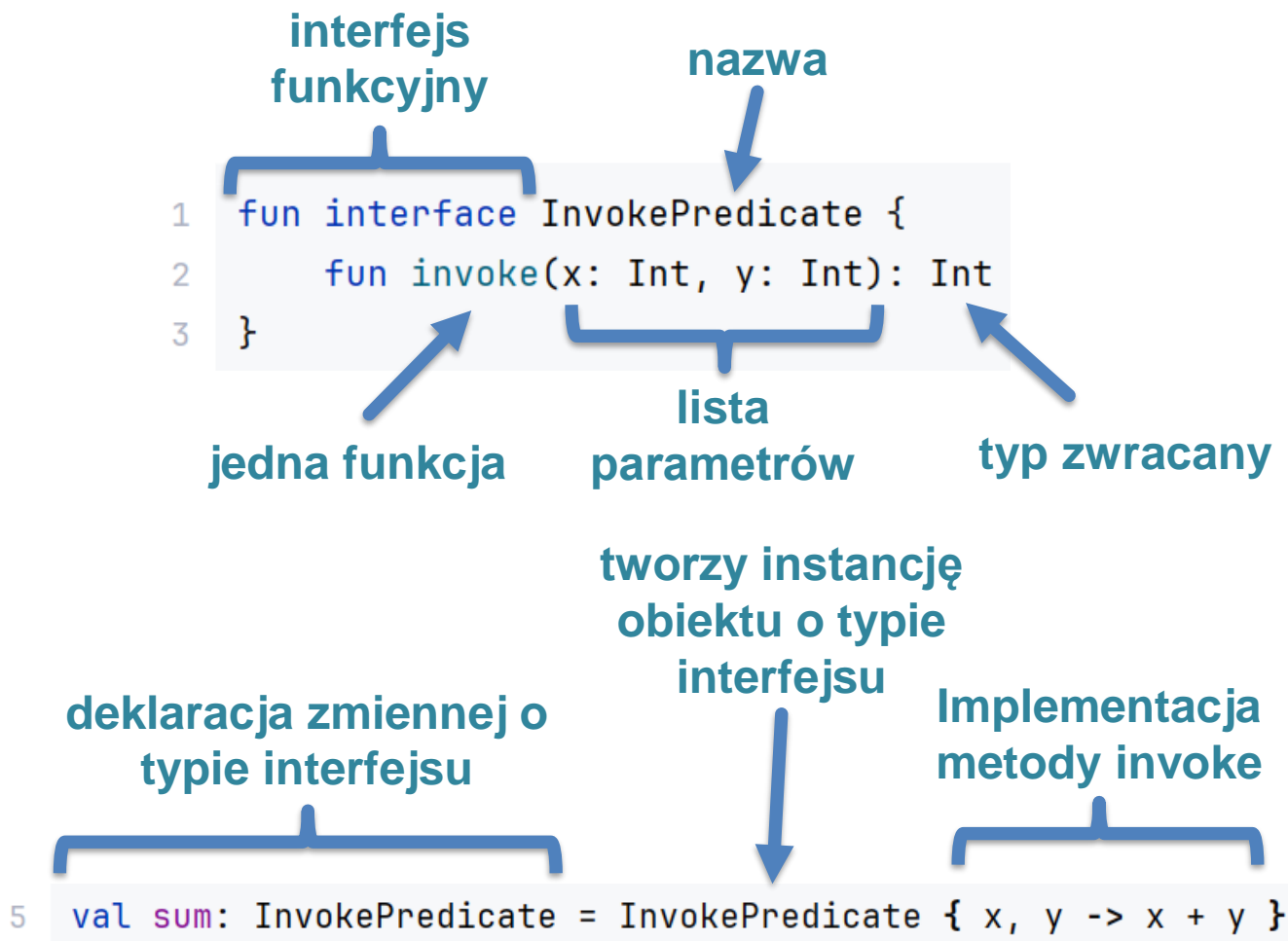

Interfejs funkcyjny

Zawierają **tylko jedną metodę**, są one wykorzystywane głównie do reprezentowania **wyrażeń lambda** jako **typów danych**.



Interfejs funkcyjny

Zawierają **tylko jedną metodę**, są one wykorzystywane głównie do reprezentowania **wyrażeń lambda** jako **typów danych**.



Single Abstract Method

Jeśli interfejs ma **dokładnie jedną metodę**, Kotlin pozwala na jego zaimplementowanie za pomocą **wyrażenia lambda**.

Bez konstruktora SAM – konieczność jawnego utworzenia obiektu o typie interfejsu

```
1 fun interface IntPredicate {  
2     fun accept(i: Int): Boolean  
3 }  
4  
5 fun isEven(): IntPredicate {  
6     return object : IntPredicate {  
7         override fun accept(i: Int): Boolean {  
8             return i % 2 == 0  
9         }  
10    }  
11 }  
12  
13 println(isEven().accept(4))  
[28]  
true
```

Single Abstract Method

Jeśli interfejs ma **dokładnie jedną metodę**, Kotlin pozwala na jego zaimplementowanie za pomocą **wyrażenia lambda**.

```
1 fun interface IntPredicate {  
2     fun accept(i: Int): Boolean  
3 }  
4  
5 val isEven = IntPredicate { it % 2 == 0 }  
6  
7 println(isEven.accept(5))  
[27]  
  
false
```

```
1 fun interface IntPredicate {  
2     fun accept(i: Int): Boolean  
3 }  
4  
5 fun isEven(): IntPredicate {  
6     return object : IntPredicate {  
7         override fun accept(i: Int): Boolean {  
8             return i % 2 == 0  
9         }  
10    }  
11 }  
12  
13 println(isEven().accept(4))  
[28]  
  
true
```

```
1 fun interface IntPredicate {  
2     fun accept(i: Int): Boolean  
3 }  
4  
5 val isEven = IntPredicate { it % 2 == 0 }  
6  
7 println(isEven.accept(5))  
[27]  
  
false
```

```
1 fun interface IntPredicate {  
2     fun accept(i: Int): Boolean  
3 }  
4  
5 fun isEven() = IntPredicate { it % 2 == 0 }  
6  
7 println(isEven().accept(5))  
✓ [3] 110ms  
  
false
```

```
1 fun interface IntPredicate {  
2     fun accept(i: Int): Boolean  
3 }  
4  
5 val isEven = IntPredicate { it % 2 == 0 }  
6  
7 println(isEven.accept(5))  
[27]
```

false

Tworzy **jedną instancję** obiektu o typie interfejsu funkcyjnego i przypisuje ją do zmiennej

```
1 fun interface IntPredicate {  
2     fun accept(i: Int): Boolean  
3 }  
4  
5 fun isEven() = IntPredicate { it % 2 == 0 }  
6  
7 println(isEven().accept(5))  
✓ [3] 110ms
```

false

Definiuje funkcję, która zwraca **nową instancję** obiektu o typie interfejsu funkcyjnego **przy każdym wywołaniu**.

Konstruktor SAM

```
1 fun interface IntPredicate {
2     fun accept(i: Int): Boolean
3 }
4
5 val isEven = IntPredicate { it % 2 == 0 }
6
7 println(isEven.accept(5))
[27]
```

false

Tworzy **jedną instancję** obiektu o typie interfejsu funkcyjnego i przypisuje ją do zmiennej

```
1 fun interface IntPredicate {
2     fun accept(i: Int): Boolean
3 }
4
5 fun isEven() = IntPredicate { it % 2 == 0 }
6
7 println(isEven().accept(5))
✓ [3] 110ms
```

false

Definiuje funkcję, która zwraca **nową instancję** obiektu o typie interfejsu funkcyjnego **przy każdym wywołaniu**.

Aspekt	val	fun
Tworzenie instancji	jednorazowe	Przy każdym wywołaniu
Wydajność	↑	↓
Zastosowanie	Stałe reguły	Dynamiczne tworzenie nowych obiektów

Funkcja przyjmuje **parametr** *threshold*, więc możliwe jest tworzenie instancji o różnych progach.

Każde wywołanie funkcji zwraca **nowy obiekt**, który działa **niezależnie od innych**

```
1 fun createThresholdPredicate(threshold: Int): IntPredicate {  
2     return IntPredicate { it > threshold }  
3 }  
4  
5  
6  
7  
8 val isGreaterThan5 = createThresholdPredicate(5)  
9 val isGreaterThan10 = createThresholdPredicate(10)  
10  
11  
12 println(isGreaterThan5.accept(7))  
13 println(isGreaterThan5.accept(4))  
14  
15 ✓ [4] 215ms  
16  
17 true  
18 false
```


Klasy abstrakcyjne vs interfejsy

Interfejsy i klasy abstrakcyjne **mogą posiadać funkcje**. Po nadpisaniu możemy uzyskać dostęp do domyślnej implementacji przez słowo kluczowe **super**.

```
7 ✓ abstract class intC {  
8 ✓     open fun printMeTo() {  
9         println("klasa")  
10     }  
11 }
```

```
1 ✓ interface intB {  
2 ✓     fun printMe() {  
3         println("interfejs")  
4     }  
5 }
```

Klasy abstrakcyjne vs interfejsy

Interfejsy i klasy abstrakcyjne **mogą posiadać funkcje**. Po nadpisaniu możemy uzyskać dostęp do domyślnej implementacji przez słowo kluczowe **super**.

```
7 abstract class intC {  
8     open fun printMeTo() {  
9         println("klasa")  
10    }  
11 }
```

```
1 interface intB {  
2     fun printMe() {  
3         println("interfejs")  
4     }  
5 }
```

implementuje
klasa interfejs
rozszerza
klasę abstrakcyjną

```
13 class A: intB, intC(){  
14     override final fun printMe(){  
15         print("Nadpisana ")  
16         super.printMe()  
17     }  
18  
19     override fun printMeTo() {  
20         print("Nadpisana ")  
21         super.printMeTo()  
22     }  
23 }
```

Dostęp do **domyślnej**
implementacji

Domyślnie **final**

interfejsy i klasy abstrakcyjne mogą posiadać **właściwości**

```
1 interface intB { val name: String }
2
3 class B: intB { override val name: String = "Rafał" }
4 val v = B()
5 v.name
[35]
```

Rafał

```
1 abstract class intB { abstract val name: String }
2
3 class B: intB() { override val name: String = "Rafał" }
4 val v = B()
5 v.name
[39]
```

Rafał

Klasy abstrakcyjne vs interfejsy

interfejs **nie może przechowywać stanu** - za wyjątkiem przypadku kiedy może to zrobić (**zła praktyka**) - przykładowo w **companion object**

Właściwość z **getterem i setterem**

Obiekt stowarzyszony

Mapa przechwująca referencje do **wszystkich obiektów o typie interfejsu**

```
1 interface intC {  
2     var name: String  
3         get() = names[this] ?: "Default name"  
4         set(value) { names[this] = value }  
5  
6     companion object {  
7         val names = mutableMapOf<Any, String>()  
8     }  
9 }
```

Klasy abstrakcyjne vs interfejsy

interfejs **nie może przechowywać stanu** - za wyjątkiem przypadku kiedy może to zrobić (**zła praktyka**) - przykładowo w **companion object**

Właściwość z **getterem i setterem**

Obiekt stowarzyszony

Mapa przechwująca referencje do **wszystkich obiektów o typie interfejsu**

Możemy odwołać się do **instancji klasy implementującej** przez słowo kluczowe **this**

```
1 interface intC {  
2     var name: String  
3     get() = names[this] ? "Default name"  
4     set(value) { names[this] = value }  
5  
6     companion object {  
7         val names = mutableMapOf<Any, String>()  
8     }  
9 }
```

Klasy abstrakcyjne vs interfejsy

interfejs **nie może przechowywać stanu** - za wyjątkiem przypadku kiedy może to zrobić (**zła praktyka**) - przykładowo w **companion object**

Właściwość z **getterem i setterem**

Obiekt stowarzyszony

Mapa przechwytująca referencje do **wszystkich obiektów o typie interfejsu**

Możemy odwołać się do **instancji klasy implementującej** przez słowo kluczowe **this**

```
1 interface intC {  
2     var name: String  
3     get() = names[this] ? "Default name"  
4     set(value) { names[this] = value }  
5  
6     companion object {  
7         val names = mutableMapOf<Any, String>()  
8     }  
9 }
```

```
11 class CA: intC  
12 class CB: intC  
13  
14 val ca = CA()  
15 ca.name = "Rafał"  
16  
17 val cb = CB()  
18 cb.name = "Radek"  
19  
20 intC.names
```

Definiujemy dwie klasy implementujące interfejs

Klasy abstrakcyjne vs interfejsy

interfejs **nie może przechowywać stanu** - za wyjątkiem przypadku kiedy może to zrobić (**zła praktyka**) - przykładowo w **companion object**

Właściwość z **getterem i setterem**

Obiekt stowarzyszony

Mapa przechwytująca referencje do **wszystkich obiektów o typie interfejsu**

Możemy odwołać się do **instancji klasy implementującej** przez słowo kluczowe **this**

```
1 interface intC {  
2     var name: String  
3     get() = names[this] ? "Default name"  
4     set(value) { names[this] = value }  
5  
6     companion object {  
7         val names = mutableMapOf<Any, String>()  
8     }  
9 }
```

```
11 class CA: intC  
12 class CB: intC  
13  
14 val ca = CA()  
15 ca.name = "Rafał"  
16  
17 val cb = CB()  
18 cb.name = "Radek"  
19  
20 intC.names
```

Definiujemy dwie klasy implementujące interfejs

Tworzymy instancje i modyfikujemy pole **names** zdefiniowane w interfejsie (**wywołujemy setter**)

Klasy abstrakcyjne vs interfejsy

interfejs **nie może przechowywać stanu** - za wyjątkiem przypadku kiedy może to zrobić (**zła praktyka**) - przykładowo w **companion object**

Właściwość z **getterem i setterem**

Obiekt stowarzyszony

Mapa przechwytująca referencje do **wszystkich obiektów o typie interfejsu**

Możemy odwołać się do **instancji klasy implementującej** przez słowo kluczowe **this**

```
1 interface intC {  
2     var name: String  
3     get() = names[this] ? "Default name"  
4     set(value) { names[this] = value }  
5  
6     companion object {  
7         val names = mutableMapOf<Any, String>()  
8     }  
9 }
```

```
11 class CA: intC  
12 class CB: intC  
13  
14 val ca = CA()  
15 ca.name = "Rafał"  
16  
17 val cb = CB()  
18 cb.name = "Radek"  
19  
20 intC.names
```

Definiujemy dwie klasy implementujące interfejs

Tworzymy instancje i modyfikujemy pole **names** zdefiniowane w interfejsie (**wywołujemy setter**)

Wywołujemy **getter** pola **names**

Klasy abstrakcyjne vs interfejsy

interfejs **nie może przechowywać stanu** - za wyjątkiem przypadku kiedy może to zrobić (**zła praktyka**) - przykładowo w **companion object**

Właściwość z **getterem i setterem**

Obiekt stowarzyszony

Mapa przechwytująca referencje do **wszystkich obiektów o typie interfejsu**

Możemy odwołać się do **instancji klasy implementującej** przez słowo kluczowe **this**


```
1 interface intC {  
2     var name: String  
3     get() = names[this] ? "Default name"  
4     set(value) { names[this] = value }  
5  
6     companion object {  
7         val names = mutableMapOf<Any, String>()  
8     }  
9 }
```

```
11 class CA: intC  
12 class CB: intC  
13  
14 val ca = CA()  
15 ca.name = "Rafał"  
16  
17 val cb = CB()  
18 cb.name = "Radek"  
19  
20 intC.names
```

{Line_48_jupyter\$CA@691b12f=Rafał, Line_48_jupyter\$CB@2f561e15=Radek}

klasa abstrakcyjna **może posiadać konstruktor**

konstruktor



```
1 ✓ abstract class Animal(val name: String, val age: Int) {  
2 ✓     fun introduce() {  
3         println("I am a $name, $age years old.")  
4     }  
5 }
```

Klasy abstrakcyjne vs interfejsy

klasa abstrakcyjna **może posiadać konstruktor**

konstruktor

```
1 ✓ abstract class Animal(val name: String, val age: Int) {  
2 ✓     fun introduce() {  
3         println("I am a $name, $age years old.")  
4     }  
5 }
```

**Brak utworzenia pola name –
tylko przekazanie do
konstruktora klasy bazowej**

konstruktor
klasy pochodnej

wywołanie
konstruktora klasy
bazowej

```
7 ✓ class Dog(name: String, age: Int, val breed: String) : Animal(name, age) {  
8 ✓     fun bark() {  
9         println("Woof! I am a $breed.")  
10     }  
11 }
```

Klasy abstrakcyjne vs interfejsy

Klasa może implementować **wiele interfejsów** i rozszerzać **tylko jedną klasę**

Definicja
interfejsów i klasy

```
1 interface Flyable { fun fly() }
2 interface Swimbable { fun swim() }
3 abstract class Animal() { abstract fun eat() }
```

Klasa
implementująca
dwa interfejsy i
rozszerzająca
jedną klasę

```
5 class Duck(val name: String) : Animal(), Flyable, Swimbable {
6     override fun fly() { println("$name is flapping its wings and flying") }
7     override fun swim() { println("$name is swimming gracefully") }
8     override fun eat() { println("$name is eating viciously") }
9 }
```