

Stefan Sokołowski

PROGRAMOWANIE OBIEKTOWE II

wykład 4 i 5

Inst. Informatyki Stosowanej, ANS

Elbląg, 2024/2025

Wywołanie zwrotne — *callback*

Wywołanie zwrotne — *callback*

Wiele klas bibliotecznych wymaga, żeby programista dostarczył im funkcji (lub klasy), którą one zastosują.

Wywołanie zwrotne — *callback*

Wiele klas bibliotecznych wymaga, żeby programista dostarczył im funkcji (lub klasy), którą one zastosują. Programista robi to przez zaimplementowanie interfejsu bibliotecznego.

Wywołanie zwrotne — *callback*

Wiele klas bibliotecznych wymaga, żeby programista dostarczył im funkcji (lub klasy), którą one zastosują. Programista robi to przez zaimplementowanie interfejsu bibliotecznego.

Klasa biblioteczna czeka na funkcję (lub klasę), którą poda programista.

Wywołanie zwrotne — *callback*

Wiele klas bibliotecznych wymaga, żeby programista dostarczył im funkcji (lub klasy), którą one zastosują. Programista robi to przez zaimplementowanie interfejsu bibliotecznego.

Klasa biblioteczna czeka na funkcję (lub klasę), którą poda programista.

BIBLIOTEKA

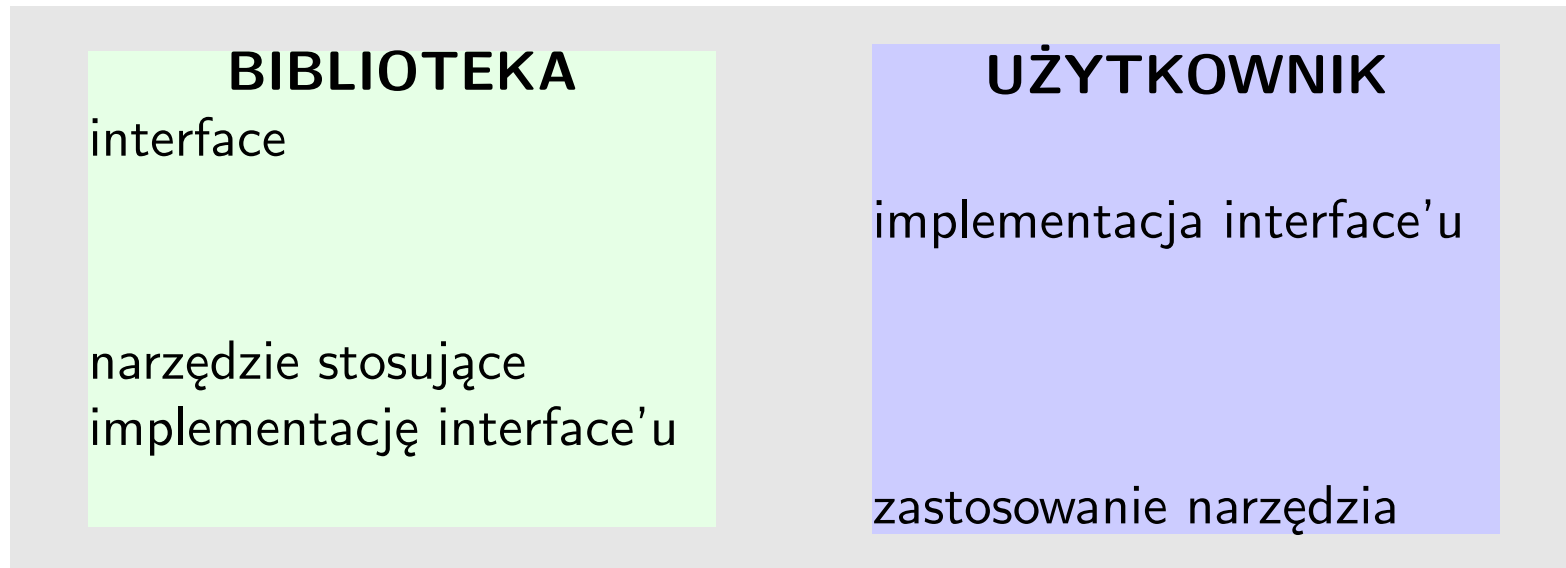
interface

narzędzie stosujące
implementację interface'u

Wywołanie zwrotne — *callback*

Wiele klas bibliotecznych wymaga, żeby programista dostarczył im funkcji (lub klasy), którą one zastosują. Programista robi to przez zaimplementowanie interfejsu bibliotecznego.

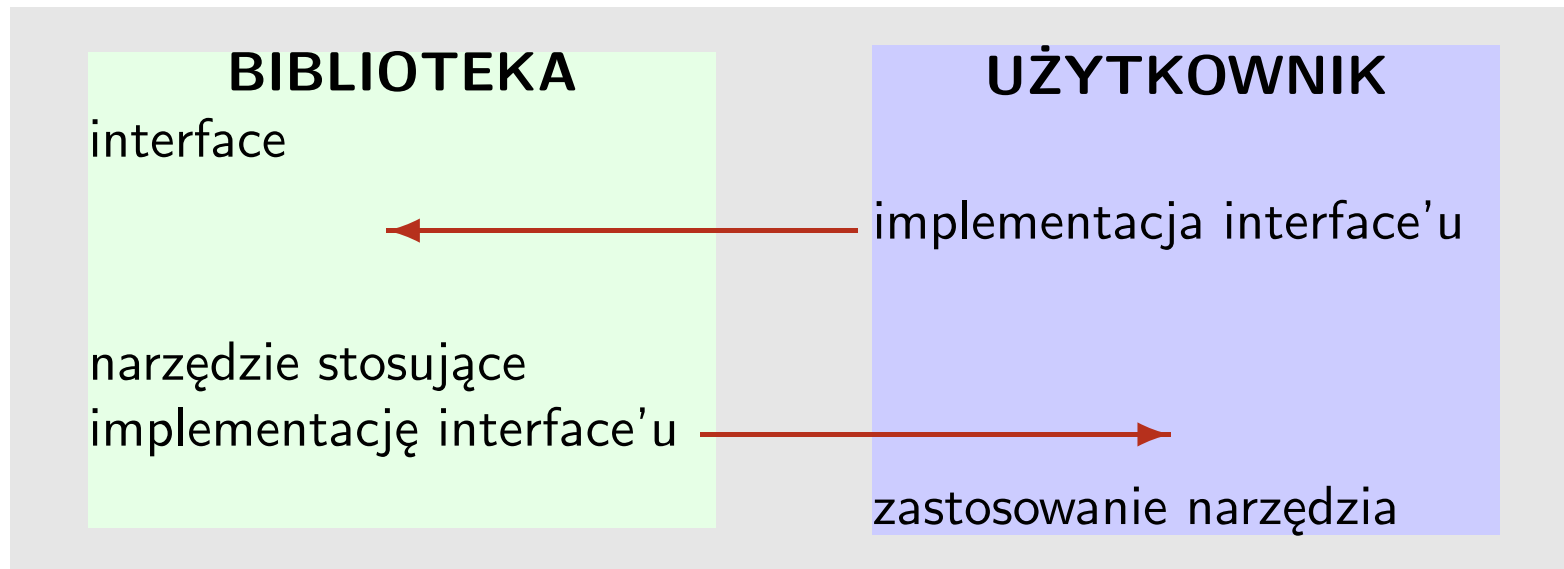
Klasa biblioteczna czeka na funkcję (lub klasę), którą poda programista.



Wywołanie zwrotne — *callback*

Wiele klas bibliotecznych wymaga, żeby programista dostarczył im funkcji (lub klasy), którą one zastosują. Programista robi to przez zaimplementowanie interfejsu bibliotecznego.

Klasa biblioteczna czeka na funkcję (lub klasę), którą poda programista.



Wywołanie zwrotne — *callback*

Przykład (trochę sztuczny, ale działa)

Wywołanie zwrotne — *callback*

Przykład (trochę sztuczny, ale działa):

W „bibliotece” znajdujemy interfejs

```
interface Zastap { String zz(char z); }
```

oraz klasę `String1` stanowiącą rozszerzenie klasy `String` o metodę

```
public String1 zamien(Zastap zast)
```

zastępującą znaki napisami w sposób wskazany w konkretnej implementacji.

Wywołanie zwrotne — *callback*

Przykład (trochę sztuczny, ale działa):

W „bibliotece” znajdujemy interfejs

```
interface Zastap { String zz(char z); }
```

oraz klasę `String1` stanowiącą rozszerzenie klasy `String` o metodę

```
public String1 zamien(Zastap zast)
```

zastępującą znaki napisami w sposób wskazany w konkretnej implementacji.

Programista dostarcza implementacji

Wywołanie zwrotne — *callback*

Przykład (trochę sztuczny, ale działa):

W „bibliotece” znajdujemy interfejs

```
interface Zastap { String zz(char z); }
```

oraz klasę `String1` stanowiącą rozszerzenie klasy `String` o metodę

```
public String1 zamien(Zastap zast)
```

zastępującą znaki napisami w sposób wskazany w konkretnej implementacji.

Programista dostarcza implementacji:

```
// opuszczanie 'a':  
class ZZ implements Zastap {  
    public String zz(char z) {  
        if (z == 'a') return "";  
        else return ""+z;  
    }  
}
```

Wywołanie zwrotne — *callback*

Przykład (trochę sztuczny, ale działa):

W „bibliotece” znajdujemy interfejs

```
interface Zastap { String zz(char z); }
```

oraz klasę `String1` stanowiącą rozszerzenie klasy `String` o metodę

```
public String1 zamien(Zastap zast)
```

zastępującą znaki napisami w sposób wskazany w konkretnej implementacji.

Programista dostarcza implementacji:

```
// opuszczanie 'a':  
class ZZ implements Zastap {  
    public String zz(char z) {  
        if (z == 'a') return "";  
        else return ""+z;  
    }  
}
```

```
// zastepowanie 'ą' przez "/a":  
class ZZ implements Zastap {  
    public String zz(char z) {  
        if (z == 'ą') return "/a";  
        else return ""+z;  
    }  
}
```

Wywołanie zwrotne — *callback*

Przykład (trochę sztuczny, ale działa):

W „bibliotece” znajdujemy interfejs

```
interface Zastap { String zz(char z); }
```

oraz klasę `String1` stanowiącą rozszerzenie klasy `String` o metodę

```
public String1 zamien(Zastap zast)
```

zastępującą znaki napisami w sposób wskazany w konkretnej implementacji.

Programista dostarcza implementacji:

```
// opuszczanie 'a':  
class ZZ implements Zastap {  
    public String zz(char z) {  
        if (z == 'a') return "";  
        else return ""+z;  
    }  
}
```

```
// zastępowanie 'ą' przez "/a":  
class ZZ implements Zastap {  
    public String zz(char z) {  
        if (z == 'ą') return "/a";  
        else return ""+z;  
    }  
}
```

wtedy komenda `new String1("dragal").zamien(new ZZ())`

Wywołanie zwrotne — *callback*

Przykład (trochę sztuczny, ale działa):

W „bibliotece” znajdujemy interfejs

```
interface Zastap { String zz(char z); }
```

oraz klasę `String1` stanowiącą rozszerzenie klasy `String` o metodę

```
public String1 zamien(Zastap zast)
```

zastępującą znaki napisami w sposób wskazany w konkretnej implementacji.

Programista dostarcza implementacji:

```
// opuszczanie 'a':
class ZZ implements Zastap {
    public String zz(char z) {
        if (z == 'a') return "";
        else return ""+z;
    }
}
```

```
// zastępowanie 'ą' przez "/a":
class ZZ implements Zastap {
    public String zz(char z) {
        if (z == 'ą') return "/a";
        else return ""+z;
    }
}
```

wtedy komenda `new String1("drągal").zamien(new ZZ())` daje

`drągl`

`dr/agal`

Wywołanie zwrotne — *callback*

Przykład (trochę sztuczny, ale działa):

W „bibliotece” znajdujemy interfejs

```
interface Zastap { String zz(char z); }
```

oraz klasę `String1` stanowiącą rozszerzenie klasy `String` o metodę

```
public String1 zamien(Zastap zast)
```

zastępującą znaki napisami w sposób wskazany w konkretnej implementacji.

Programista dostarcza implementacji:

```
// opuszczanie 'a':
class ZZ implements Zastap {
    public String zz(char z) {
        if (z == 'a') return "";
        else return ""+z;
    }
}
```

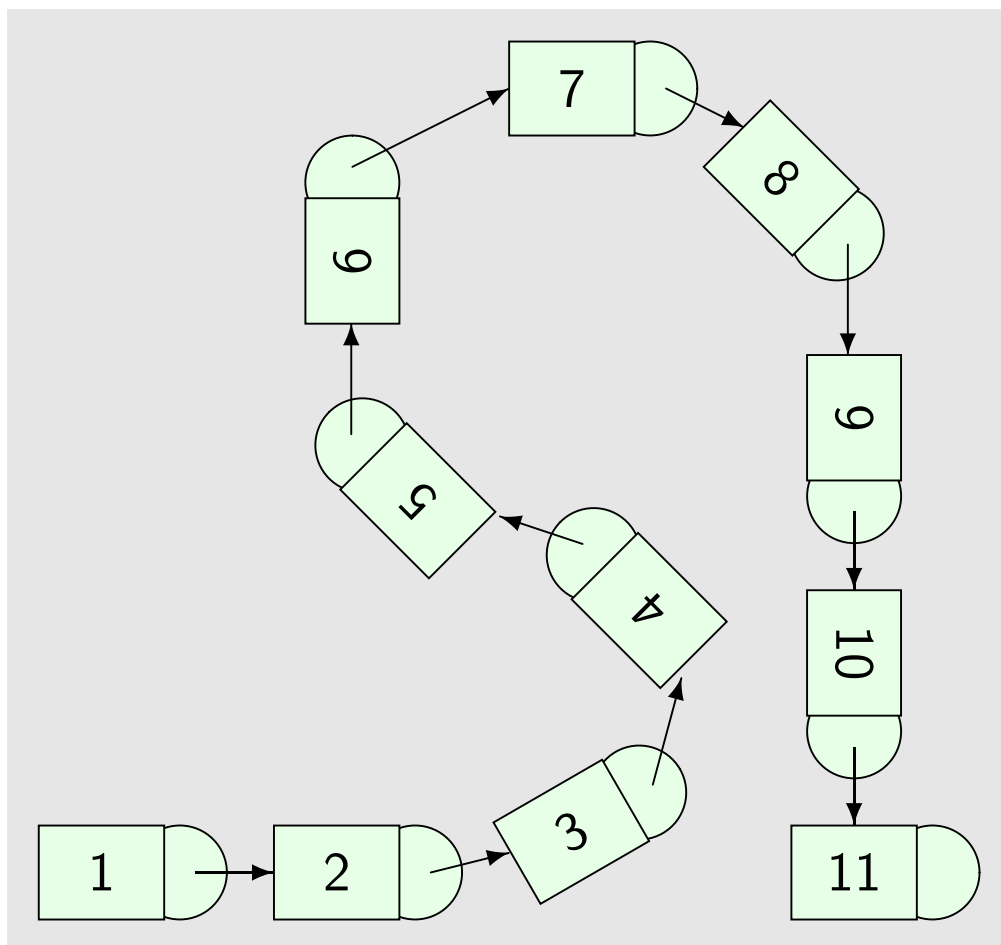
```
// zastępowanie 'ą' przez "/a":
class ZZ implements Zastap {
    public String zz(char z) {
        if (z == 'ą') return "/a";
        else return ""+z;
    }
}
```

wtedy komenda `new String1("dragal").zamien(new ZZ())` daje

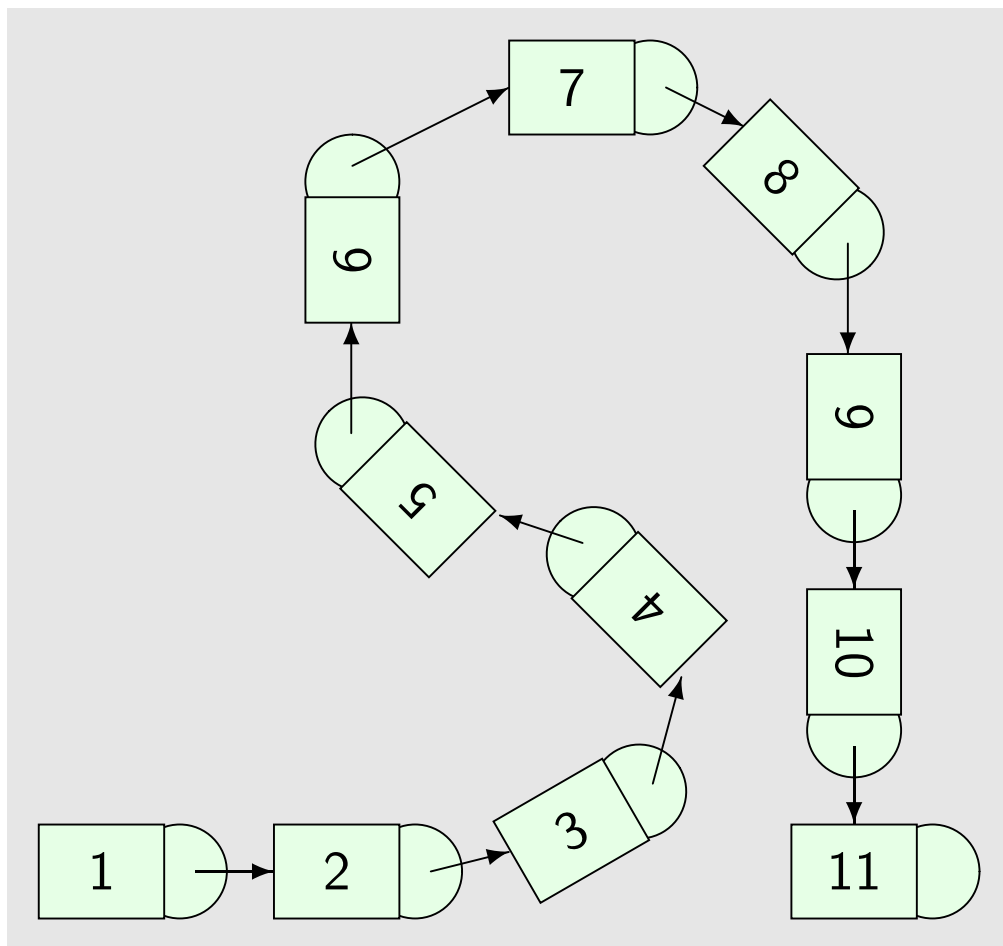
`dragl`

`dr/agal`

Co zamiast wskaźników?

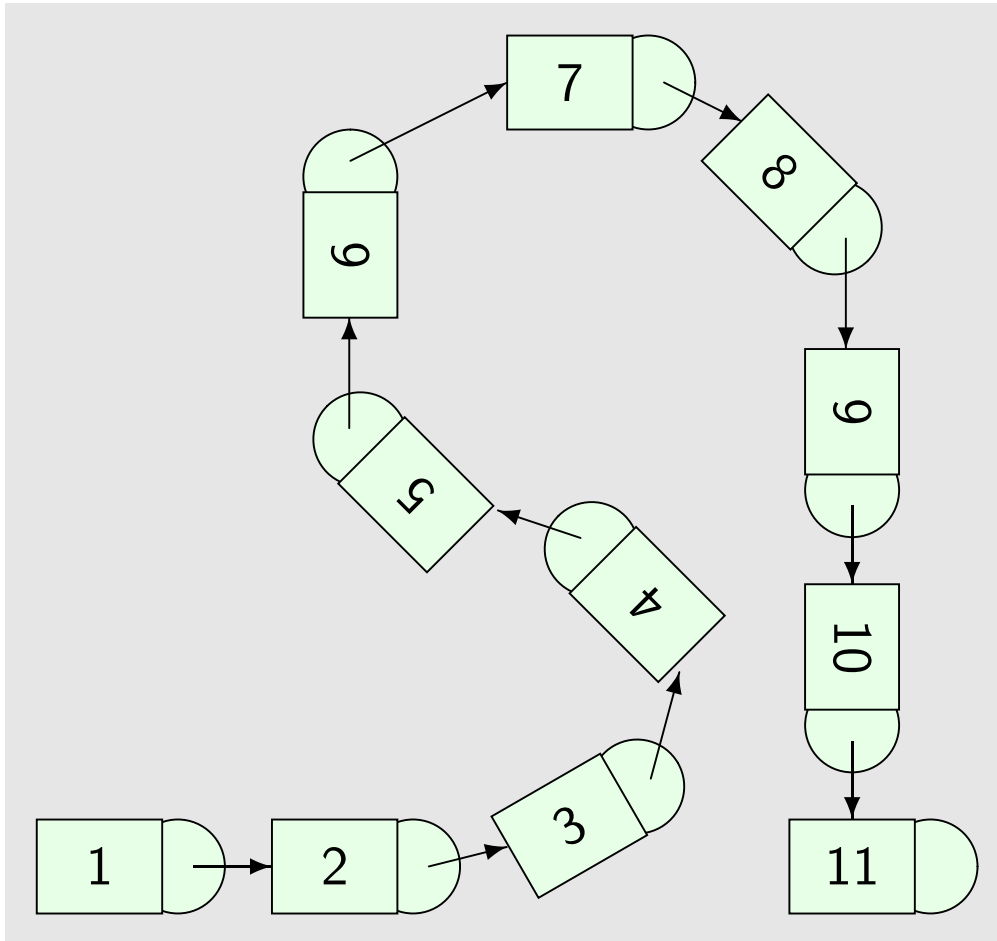


Co zamiast wskaźników?



W Javie **nie** ma **jawnych** wskaźników.

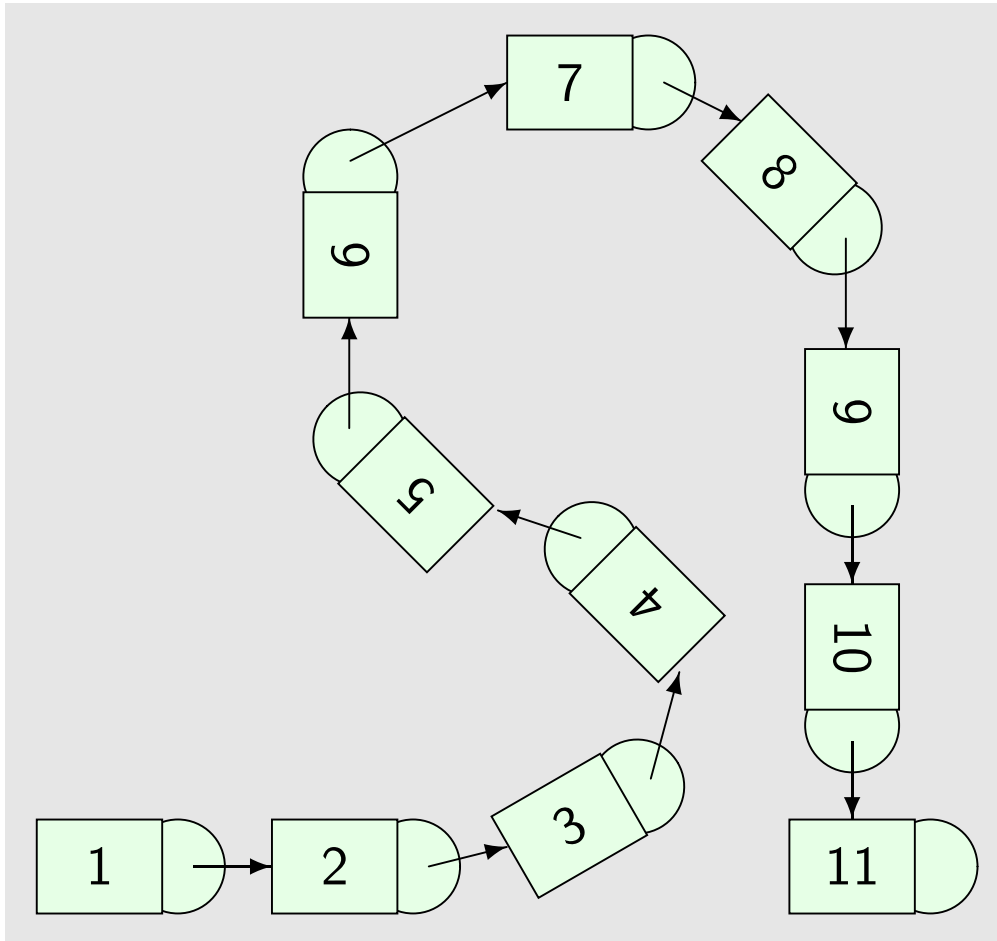
Co zamiast wskaźników?



W Javie **nie** ma **jawnych** wskaźników.

Jak więc zaimplementować struktury dynamiczne, takie jak listy lub drzewa?

Co zamiast wskaźników?



W Javie **nie ma jawnych** wskaźników.

Jak więc zaimplementować struktury dynamiczne, takie jak listy lub drzewa?

zamiast **typów wskaźnikowych**
klasy rekursywne

Co zamiast wskaźników?

zamiast **typów wskaźnikowych** **klasy rekursywne**

Co zamiast wskaźników?

zamiast **typów wskaźnikowych** **klasy rekursywne**

Następny węzeł listy znajduje się nie „dalej” tylko „wewnątrz”.

Co zamiast wskaźników?

zamiast **typów wskaźnikowych** klasy rekursywne

Następny węzeł listy znajduje się nie „dalej” tylko „wewnątrz”.



Klasy rekursywne

```
class Wezel {  
    private int glowa;  
    private Wezel ogon;  
  
    // konstruktor:  
    public Wezel(int liczba, Wezel lista) {  
        glowa=liczba; ogon=lista;  
    }  
  
    // selektory:  
    public int glowa() { return glowa; }  
    public Wezel ogon() { return ogon; }  
}
```

Klasy rekursywne

```
class Wezel {  
    private int glowa;  
    private Wezel ogon;  
  
    // konstruktor:  
    public Wezel(int liczba, Wezel lista) {  
        glowa=liczba; ogon=lista;  
    }  
  
    // selektory:  
    public int glowa() { return glowa; }  
    public Wezel ogon() { return ogon; }  
}
```



Klasy rekursywne

```
class Wezel {  
    private int glowa;  
    private Wezel ogon;  
  
    // konstruktor:  
    public Wezel(int liczba, Wezel lista) {  
        glowa=liczba; ogon=lista;  
    }  
  
    // selektory:  
    public int glowa() { return glowa; }  
    public Wezel ogon() { return ogon; }  
}
```



Lista może być albo null-em, albo wynikiem zastosowania konstruktora

Klasy rekursywne

```
class Wezel {  
    private int glowa;  
    private Wezel ogon;  
  
    // konstruktor:  
    public Wezel(int liczba, Wezel lista) {  
        glowa=liczba; ogon=lista;  
    }  
  
    // selektory:  
    public int glowa() { return glowa; }  
    public Wezel ogon() { return ogon; }  
}
```



Lista może być albo null-em, albo wynikiem zastosowania konstruktora:

```
Wezel lista = null;
```



Klasy rekursywne

```
class Wezel {  
    private int glowa;  
    private Wezel ogon;  
  
    // konstruktor:  
    public Wezel(int liczba, Wezel lista) {  
        glowa=liczba; ogon=lista;  
    }  
  
    // selektory:  
    public int glowa() { return glowa; }  
    public Wezel ogon() { return ogon; }  
}
```



Lista może być albo null-em, albo wynikiem zastosowania konstruktora:

```
Wezel lista = null;  
lista = new Wezel(0, lista);
```

0, ☐

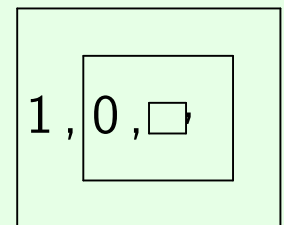
Klasy rekursywne

```
class Wezel {  
    private int glowa;  
    private Wezel ogon;  
  
    // konstruktor:  
    public Wezel(int liczba, Wezel lista) {  
        glowa=liczba; ogon=lista;  
    }  
  
    // selektory:  
    public int glowa() { return glowa; }  
    public Wezel ogon() { return ogon; }  
}
```



Lista może być albo null-em, albo wynikiem zastosowania konstruktora:

```
Wezel lista = null;  
lista = new Wezel(0, lista);  
lista = new Wezel(1, lista);
```



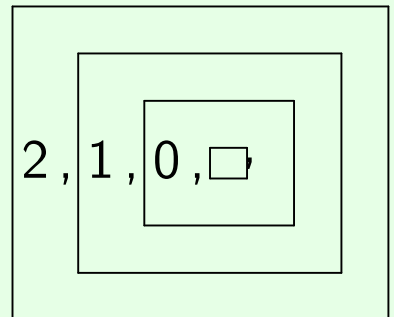
Klasy rekursywne

```
class Wezel {  
    private int glowa;  
    private Wezel ogon;  
  
    // konstruktor:  
    public Wezel(int liczba, Wezel lista) {  
        glowa=liczba; ogon=lista;  
    }  
  
    // selektory:  
    public int glowa() { return glowa; }  
    public Wezel ogon() { return ogon; }  
}
```



Lista może być albo null-em, albo wynikiem zastosowania konstruktora:

```
Wezel lista = null;  
lista = new Wezel(0, lista);  
lista = new Wezel(1, lista);  
lista = new Wezel(2, lista);
```



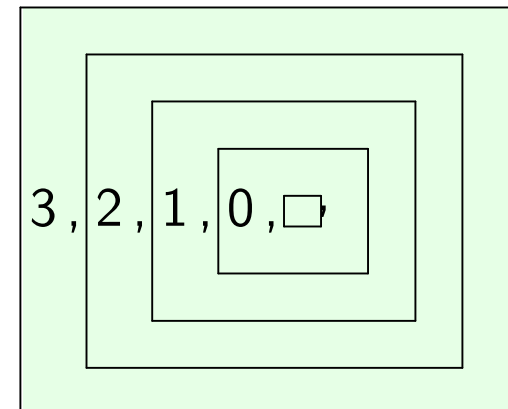
Klasy rekursywne

```
class Wezel {  
    private int glowa;  
    private Wezel ogon;  
  
    // konstruktor:  
    public Wezel(int liczba, Wezel lista) {  
        glowa=liczba; ogon=lista;  
    }  
  
    // selektory:  
    public int glowa() { return glowa; }  
    public Wezel ogon() { return ogon; }  
}
```



Lista może być albo null-em, albo wynikiem zastosowania konstruktora:

```
Wezel lista = null;  
lista = new Wezel(0, lista);  
lista = new Wezel(1, lista);  
lista = new Wezel(2, lista);  
lista = new Wezel(3, lista);
```



Klasy rekursywne

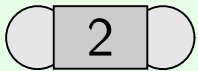
```
class Drzewo {  
    private int tresc;  
    private Drzewo lewe, prawe;  
  
    public Drzewo(int tr, Drzewo le, Drzewo pr) {  
        tresc=tr; lewe=le; prawe=pr;  
    }  
  
    public int tresc() { return tresc; }  
    public Drzewo lewe() { return lewe; }  
    public Drzewo prawe() { return prawe; }  
}
```



Klasy rekursywne

```
class Drzewo {  
    private int tresc;  
    private Drzewo lewe, prawe;  
  
    public Drzewo(int tr, Drzewo le, Drzewo pr) {  
        tresc=tr; lewe=le; prawe=pr;  
    }  
  
    public int tresc() { return tresc; }  
    public Drzewo lewe() { return lewe; }  
    public Drzewo prawe() { return prawe; }  
}
```

```
new Drzewo(2,  
    null,  
    null  
)
```



Klasy rekursywne

```
class Drzewo {
    private int tresc;
    private Drzewo lewe, prawe;

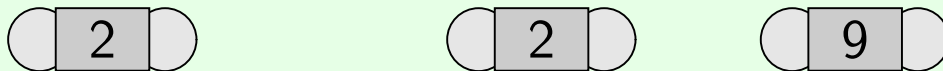
    public Drzewo(int tr, Drzewo le, Drzewo pr) {
        tresc=tr; lewe=le; prawe=pr;
    }

    public int tresc() { return tresc; }
    public Drzewo lewe() { return lewe; }
    public Drzewo prawe() { return prawe; }
}
```



```
new Drzewo(2,
    null,
    null
)
```

```
new Drzewo(2, null, null),
new Drzewo(9, null, null)
```



Klasy rekursywne

```
class Drzewo {
    private int tresc;
    private Drzewo lewe, prawe;

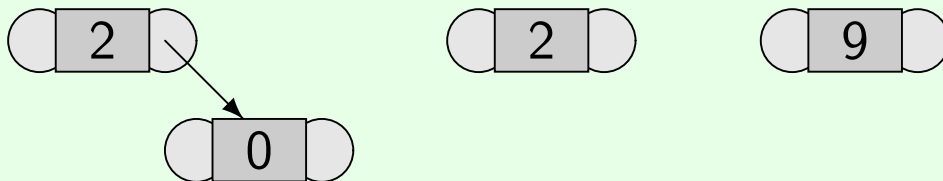
    public Drzewo(int tr, Drzewo le, Drzewo pr) {
        tresc=tr; lewe=le; prawe=pr;
    }

    public int tresc() { return tresc; }
    public Drzewo lewe() { return lewe; }
    public Drzewo prawe() { return prawe; }
}
```



```
new Drzewo(2,
    null,
    new Drzewo(0, null, null)
)
```

```
new Drzewo(2, null, null),
new Drzewo(9, null, null)
```

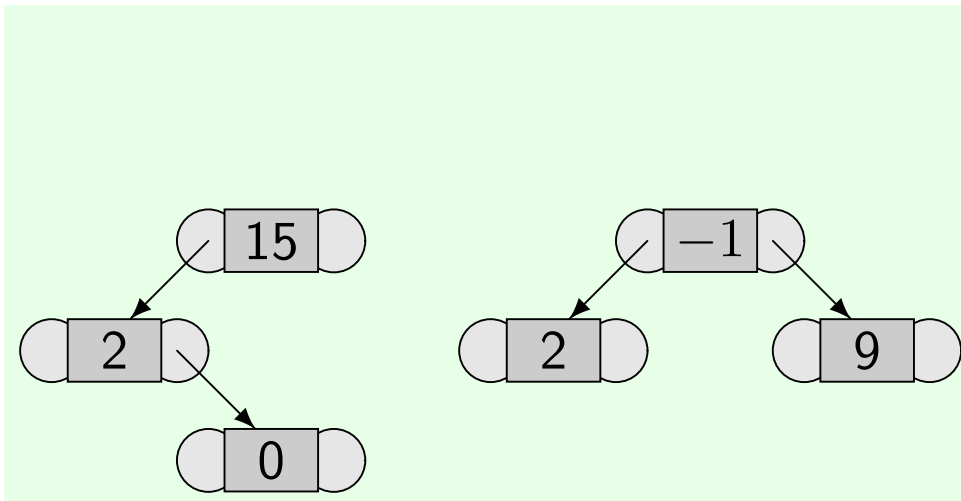


Klasy rekursywne

```
class Drzewo {
    private int tresc;
    private Drzewo lewe, prawe;

    public Drzewo(int tr, Drzewo le, Drzewo pr) {
        tresc=tr; lewe=le; prawe=pr;
    }

    public int tresc() { return tresc; }
    public Drzewo lewe() { return lewe; }
    public Drzewo prawe() { return prawe; }
}
```



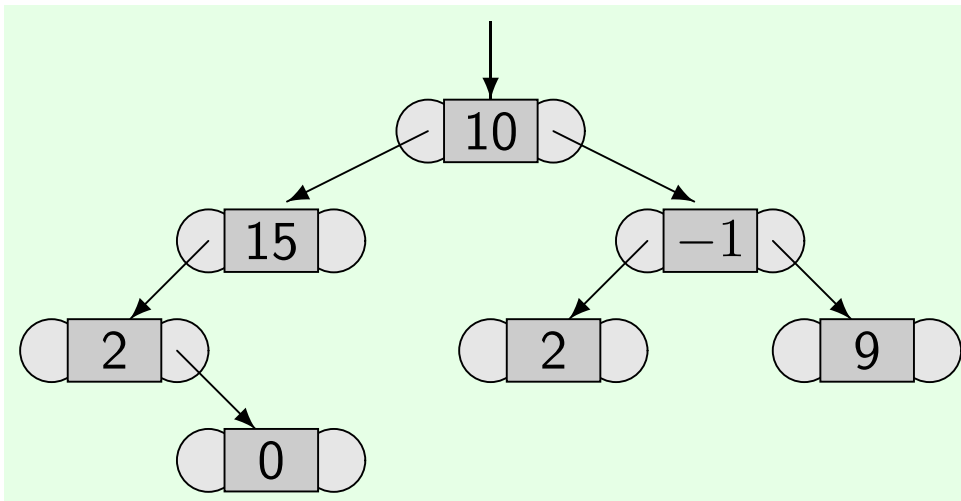
```
new Drzewo(15,
    new Drzewo(2,
        null,
        new Drzewo(0, null, null)
    ),
    null
)
new Drzewo(-1,
    new Drzewo(2, null, null),
    new Drzewo(9, null, null)
)
```

Klasy rekursywne

```
class Drzewo {
    private int tresc;
    private Drzewo lewe, prawe;

    public Drzewo(int tr, Drzewo le, Drzewo pr) {
        tresc=tr; lewe=le; prawe=pr;
    }

    public int tresc() { return tresc; }
    public Drzewo lewe() { return lewe; }
    public Drzewo prawe() { return prawe; }
}
```



```
new Drzewo(10,
    new Drzewo(15,
        new Drzewo(2,
            null,
            new Drzewo(0, null, null)
        ),
        null
    ),
    new Drzewo(-1,
        new Drzewo(2, null, null),
        new Drzewo(9, null, null)
    )
)
```

Klasy generyczne

— klasy zależne od innych klas

Klasy generyczne

— klasy zależne od innych klas; np.

generyczna	
<code>ArrayList<T></code>	

Klasy generyczne

— klasy zależne od innych klas; np.

generyczna	konkretne
ArrayList<T>	ArrayList<String> — listy napisów
	ArrayList<Double> — listy liczb rzeczywistych
	ArrayList<ArrayList<Double>> — listy list liczb rzeczywistych

Klasy generyczne

— klasy zależne od innych klas; np.

generyczna	konkretne
ArrayList<T>	ArrayList<String> — listy napisów
	ArrayList<Double> — listy liczb rzeczywistych
	ArrayList<ArrayList<Double>> — listy list liczb rzeczywistych

Klasa generyczna posiada parametr (lub parametry), za który można wstawić klasę.

Klasy generyczne

— klasy zależne od innych klas; np.

generyczna	konkretne
ArrayList<T>	ArrayList<String> — listy napisów
	ArrayList<Double> — listy liczb rzeczywistych
	ArrayList<ArrayList<Double>> — listy list liczb rzeczywistych

Klasa generyczna posiada parametr (lub parametry), za który można wstawić klasę. Ale działania niezależne od konkretnego parametru można wykonywać w postaci generycznej.

Klasy generyczne

— klasy zależne od innych klas; np.

generyczna	konkretne
ArrayList<T>	ArrayList<String> — listy napisów
	ArrayList<Double> — listy liczb rzeczywistych
	ArrayList<ArrayList<Double>> — listy list liczb rzeczywistych

Klasa generyczna posiada parametr (lub parametry), za który można wstawić klasę. Ale działania niezależne od konkretnego parametru można wykonywać w postaci generycznej.

```
class Para<T0,T1> {
    private T0 fst; private T1 snd;
    public Para(T0 x, T1 y) { fst=x; snd=y; }
    public T0 fst() { return fst; }
    public T1 snd() { return snd; }
    public String toString() {
        return fst().toString() + " " + snd().toString();
    }
}
```

Klasy generyczne

— klasy zależne od innych klas; np.

generyczna	konkretne
ArrayList<T>	ArrayList<String> — listy napisów
	ArrayList<Double> — listy liczb rzeczywistych
	ArrayList<ArrayList<Double>> — listy list liczb rzeczywistych

Klasa generyczna posiada parametr (lub parametry), za który można wstawić klasę. Ale działania niezależne od konkretnego parametru można wykonywać w postaci generycznej.

```
class Para<T0,T1> {
    private T0 fst; private T1 snd;
    public Para(T0 x, T1 y) { fst=x; snd=y; }
    public T0 fst() { return fst; }
    public T1 snd() { return snd; }
    public String toString() {
        return fst().toString() + " " + snd().toString();
    }
}
```

parametry klasy Para

Klasy generyczne

— klasy zależne od innych klas; np.

generyczna	konkretne
ArrayList<T>	ArrayList<String> — listy napisów
	ArrayList<Double> — listy liczb rzeczywistych
	ArrayList<ArrayList<Double>> — listy list liczb rzeczywistych

Klasa generyczna posiada parametr (lub parametry), za który można wstawić klasę. Ale działania niezależne od konkretnego parametru można wykonywać w postaci generycznej.

```

class Para<T0,T1> {
    private T0 fst; private T1 snd;
    public Para(T0 x, T1 y) { fst=x; snd=y; }
    public T0 fst() { return fst; }
    public T1 snd() { return snd; }
    public String toString() {
        return fst().toString() + " " + snd().toString();
    }
}

```

parametry klasy Para

konstruktor

Klasy generyczne

— klasy zależne od innych klas; np.

generyczna	konkretne
ArrayList<T>	ArrayList<String> — listy napisów
	ArrayList<Double> — listy liczb rzeczywistych
	ArrayList<ArrayList<Double>> — listy list liczb rzeczywistych

Klasa generyczna posiada parametr (lub parametry), za który można wstawić klasę. Ale działania niezależne od konkretnego parametru można wykonywać w postaci generycznej.

```
class Para<T0,T1> {
    private T0 fst; private T1 snd;
    public Para(T0 x, T1 y) { fst=x; snd=y; }
    public T0 fst() { return fst; }
    public T1 snd() { return snd; }
    public String toString() {
        return fst().toString() + " " + snd().toString();
    }
}
```

parametry klasy Para

konstruktor

selektory

Klasy generyczne

— klasy zależne od innych klas; np.

generyczna	konkretne
ArrayList<T>	ArrayList<String> — listy napisów
	ArrayList<Double> — listy liczb rzeczywistych
	ArrayList<ArrayList<Double>> — listy list liczb rzeczywistych

Klasa generyczna posiada parametr (lub parametry), za który można wstawić klasę. Ale działania niezależne od konkretnego parametru można wykonywać w postaci generycznej.

```
class Para<T0,T1> {
    private T0 fst; private T1 snd;
    public Para(T0 x, T1 y) { fst=x; snd=y; }
    public T0 fst() { return fst; }
    public T1 snd() { return snd; }
    public String toString() {
        return fst().toString() + " " + snd().toString();
    }
}
```

parametry klasy Para

konstruktor

selektory

używa metody toString
klas T0 i T1

Użycie klasy generycznej

Jednoczesne znajdowanie maximum i minimum

Użycie klasy generycznej

Jednoczesne znajdowanie maximum i minimum:

```
public static Para<Double,Double> maxMin(double[] a) {  
    var mm = new Para<Double,Double>(a[0], a[0]);  
    for (int i=1; i<a.length; i++) {  
        mm = new Para<Double,Double>(  
            (mm.fst()<a[i] ? a[i] : mm.fst()),  
            (mm.snd()>a[i] ? a[i] : mm.snd())  
        );  
    }  
    return mm;  
}
```

Użycie klasy generycznej

Jednoczesne znajdowanie maximum i minimum:

```
public static Para<Double,Double> maxMin(double[] a) {  
    var mm = new Para<Double,Double>(a[0], a[0]);  
    for (int i=1; i<a.length; i++) {  
        mm = new Para<Double,Double>(  
            (mm.fst()<a[i] ? a[i] : mm.fst()),  
            (mm.snd()>a[i] ? a[i] : mm.snd())  
        );  
    }  
    return mm;  
}
```

tymczasowe
max-min



Użycie klasy generycznej

Jednoczesne znajdowanie maximum i minimum:

```
public static Para<Double,Double> maxMin(double[] a) {  
    var mm = new Para<Double,Double>(a[0], a[0]);  
    for (int i=1; i<a.length; i++) {  
        mm = new Para<Double,Double>(  
            (mm.fst()<a[i] ? a[i] : mm.fst()),  
            (mm.snd()>a[i] ? a[i] : mm.snd())  
        );  
    }  
    return mm;  
}
```

tymczasowe
max-min

poprawianie
pary max-min

```
public static Para<Double,Double> maxMin(double[] a) {
    var mm = new Para<Double,Double>(a[0], a[0]);
    for (int i=1; i<a.length; i++) {
        mm = new Para<Double,Double>(
            (mm.fst()<a[i] ? a[i] : mm.fst()),
            (mm.snd()>a[i] ? a[i] : mm.snd())
        );
    }
    return mm;
}
```

poprawianie pary max-min

```
iis.ans-elblag.pl/~stefan/Dydaktyka/ProgObiII/Prog04/P2_Generyki/{ParaTrojka.java  
G0_MaxMinDouble.java
```

Użycie klas generycznych

Klasa **generyczna** może zależeć od więcej niż jednego parametru.

Użycie klas generycznych

Klasa **generyczna** może zależeć od więcej niż jednego parametru.

W poprzednim przykładzie:

```
class Para<T0,T1> {  
    private T0 fst; private T1 snd;  
    public Para(T0 x, T1 y) { fst=x; snd=y; }  
    public T0 fst() { return fst; }  
    public T1 snd() { return snd; }  
    public String toString() {  
        return fst().toString() + " " + snd().toString();  
    }  
}
```

Użycie klas generycznych

Klasa **generyczna** może zależeć od więcej niż jednego parametru.

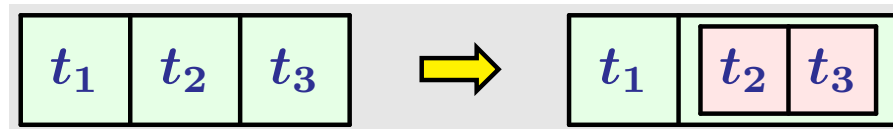
W poprzednim przykładzie:

```
class Para<T0,T1> {  
    private T0 fst; private T1 snd;  
    public Para(T0 x, T1 y) { fst=x; snd=y; }  
    public T0 fst() { return fst; }  
    public T1 snd() { return snd; }  
    public String toString() {  
        return fst().toString() + " " + snd().toString();  
    }  
}
```

Klasa **T0** może być różna od klasy **T1**.

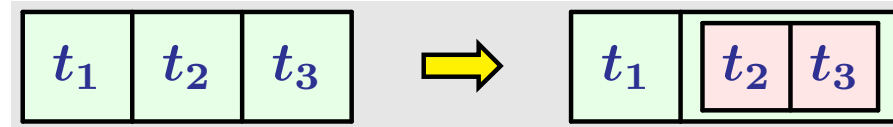
Użycie klas generycznych

Trójka (t_1, t_2, t_3) jako złożenie par $(t_1, (t_2, t_3))$:



Użycie klas generycznych

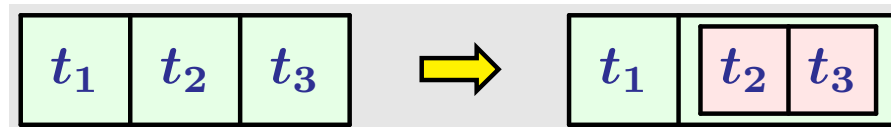
Trójka (t_1, t_2, t_3) jako złożenie par $(t_1, (t_2, t_3))$:



```
class Trojka<T0,T1,T2> extends Para<T0, Para<T1,T2>> {  
    public Trojka(T0 x, T1 y, T2 z) {  
        super(x, new Para<T1,T2>(y,z));  
    }  
    public T0 pie() { return fst(); }  
    public T1 dru() { return snd().fst(); }  
    public T2 trz() { return snd().snd(); }  
}
```

Użycie klas generycznych

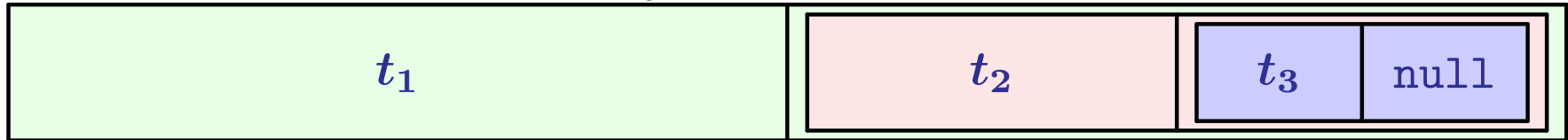
Trójka (t_1, t_2, t_3) jako złożenie par $(t_1, (t_2, t_3))$:



```
class Trojka<T0,T1,T2> extends Para<T0, Para<T1,T2>> {
    public Trojka(T0 x, T1 y, T2 z) {
        super(x, new Para<T1,T2>(y,z));
    }
    public T0 pie() { return fst(); }
    public T1 dru() { return snd().fst(); }
    public T2 trz() { return snd().snd(); }
}
```

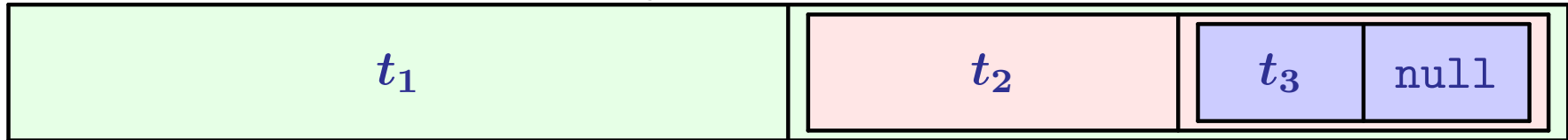
Rekursja + generyczność

Lista jako złożenie par:



Rekursja + generyczność

Lista jako złożenie par:

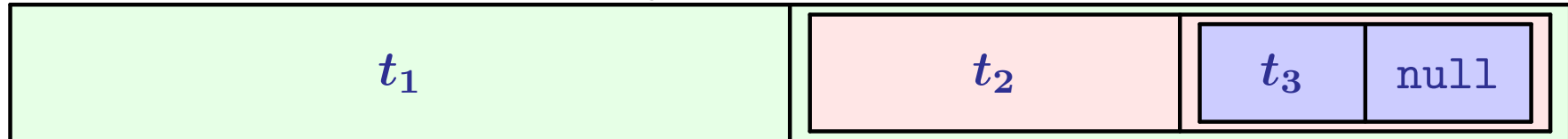


Generyczna lista własnej roboty:

```
class Lista<T> {
    private Para<T, Lista<T>> pp;
    public Lista() { pp = null; }
    public Lista(T t, Lista<T> tt) {
        pp = new Para<T,Lista<T>>(t, tt);
    }
    public T hd() { return pp.fst(); }
    public Lista<T> tl() { return pp.snd(); }
    public Boolean jestPusta() { return pp==null; }
}
```

Rekursja + generyczność

Lista jako złożenie par:



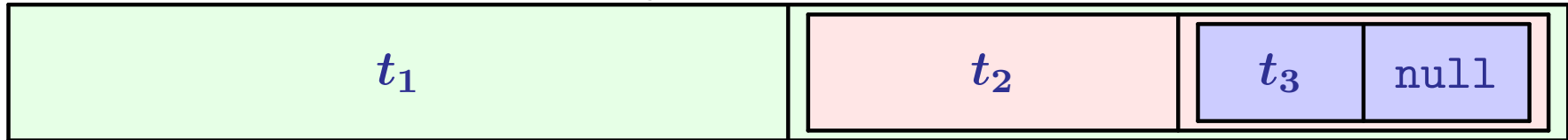
Generyczna lista własnej roboty:

```
class Lista<T> {
    private Para<T, Lista<T>> pp;
    public Lista() { pp = null; }
    public Lista(T t, Lista<T> tt) {
        pp = new Para<T,Lista<T>>(t, tt);
    }
    public T hd() { return pp.fst(); }
    public Lista<T> tl() { return pp.snd(); }
    public Boolean jestPusta() { return pp==null; }
}
```

dwa
konstruktory

Rekursja + generyczność

Lista jako złożenie par:



Generyczna lista własnej roboty:

```
class Lista<T> {
    private Para<T, Lista<T>> pp;
    public Lista() { pp = null; }
    public Lista(T t, Lista<T> tt) {
        pp = new Para<T,Lista<T>>(t, tt);
    }
```

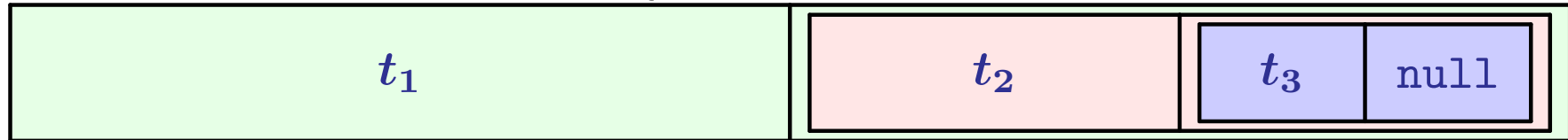
dwa
konstruktory

```
    public T hd() { return pp.fst(); }
    public Lista<T> tl() { return pp.snd(); }
    public Boolean jestPusta() { return pp==null; }
}
```

selektory

Rekursja + generyczność

Lista jako złożenie par:



Generyczna lista własnej roboty:

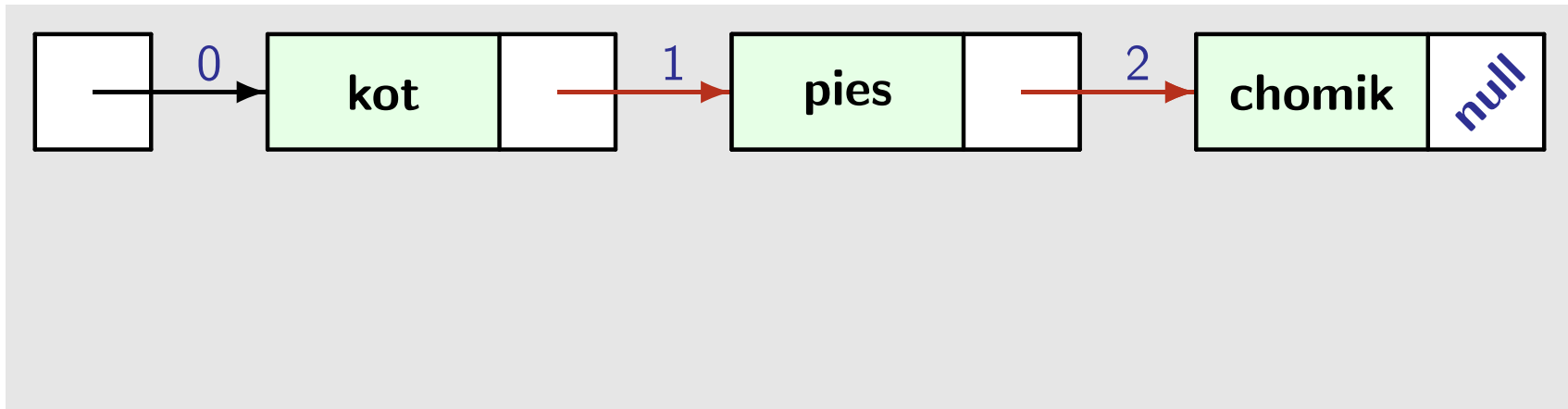
```
class Lista<T> {
    private Para<T, Lista<T>> pp;
    public Lista() { pp = null; }
    public Lista(T t, Lista<T> tt) {
        pp = new Para<T,Lista<T>>(t, tt);
    }
    public T hd() { return pp.fst(); }
    public Lista<T> tl() { return pp.snd(); }
    public Boolean jestPusta() { return pp==null; }
}
```

dwa
konstruktory

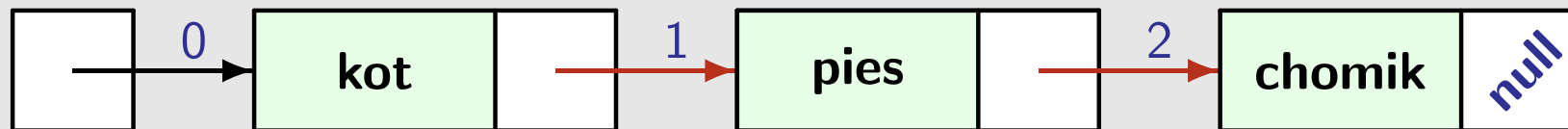
selektory

Wstawianie do wnętrza listy

Wstawianie do wnętrza listy



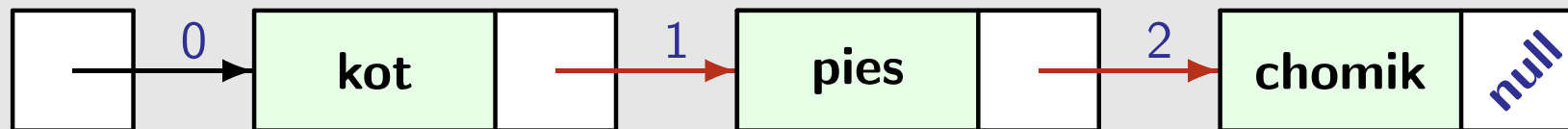
Wstawianie do wnętrza listy



wstawić
na poz. 2



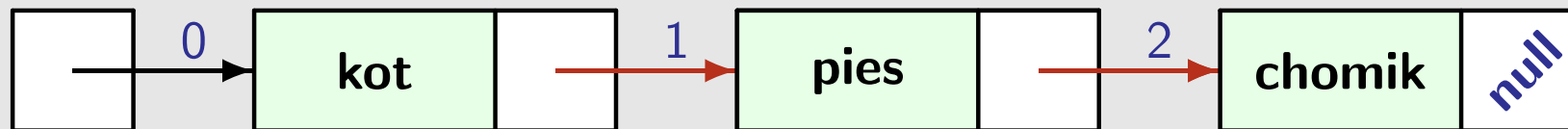
Wstawianie do wnętrza listy



wstawić
na poz. 2



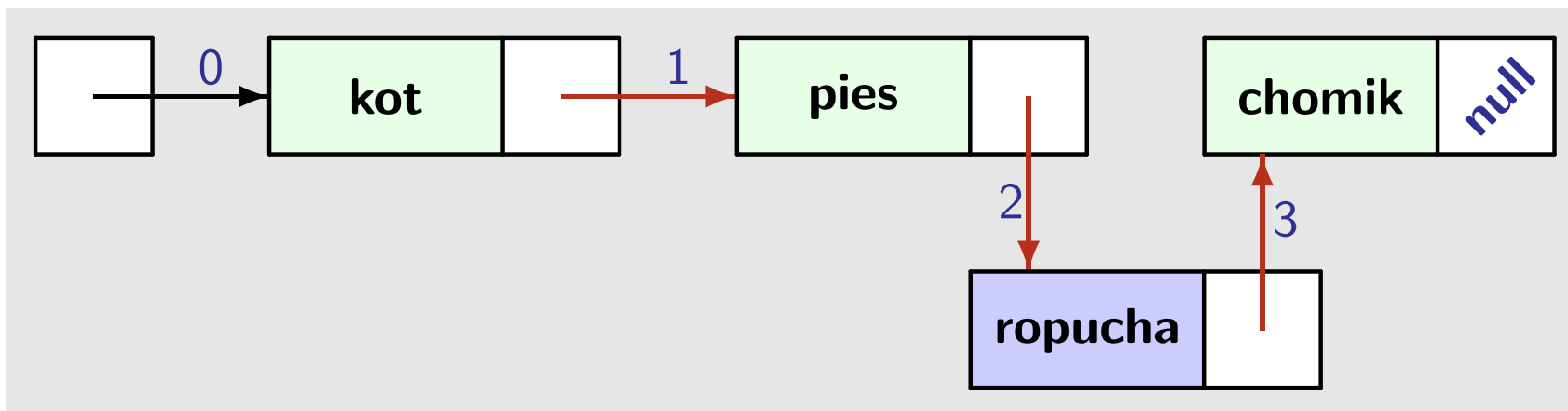
Wstawianie do wnętrza listy



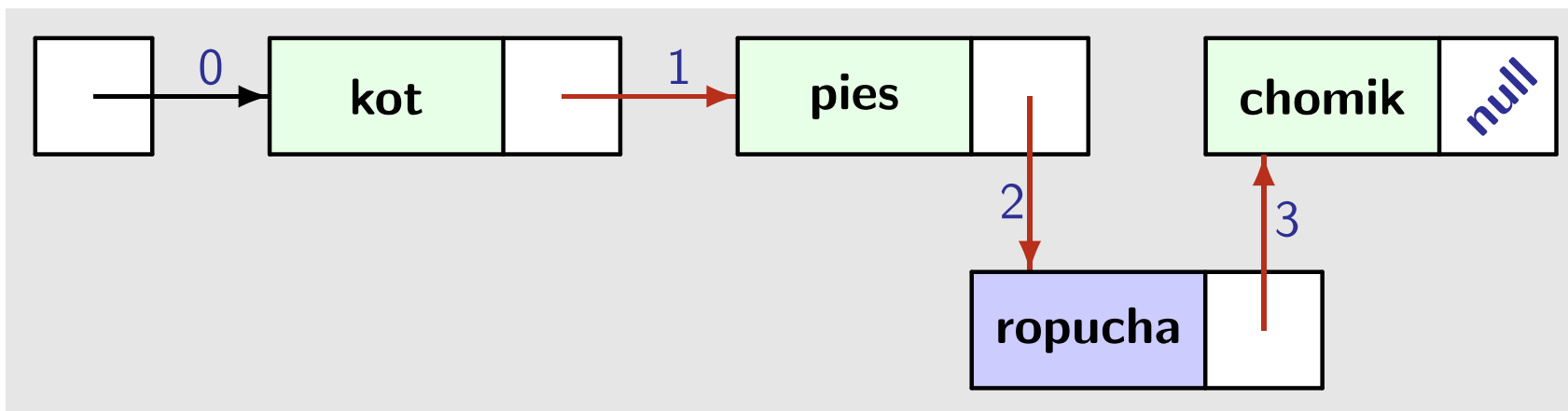
wstawić
na poz. 2



Wstawianie do wnętrza listy



Wstawianie do wnętrza listy



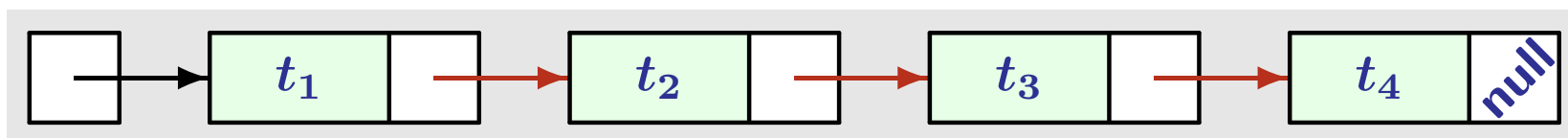
Zacyklenie listy

Przy konstruowaniu list przez pary należy
bardzo uważać na listy cykliczne!

Zacyklenie listy

Przy konstruowaniu list przez pary należy
bardzo uważać na listy cykliczne!

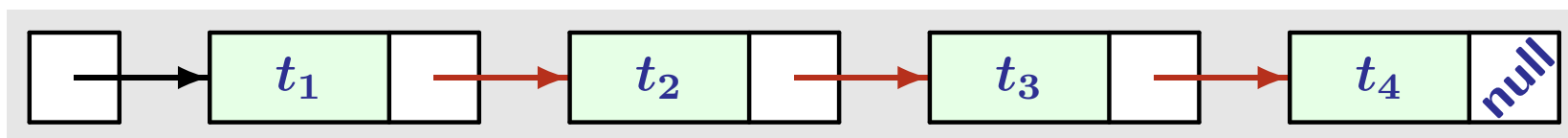
Lista jest realizowana przez wskaźniki:



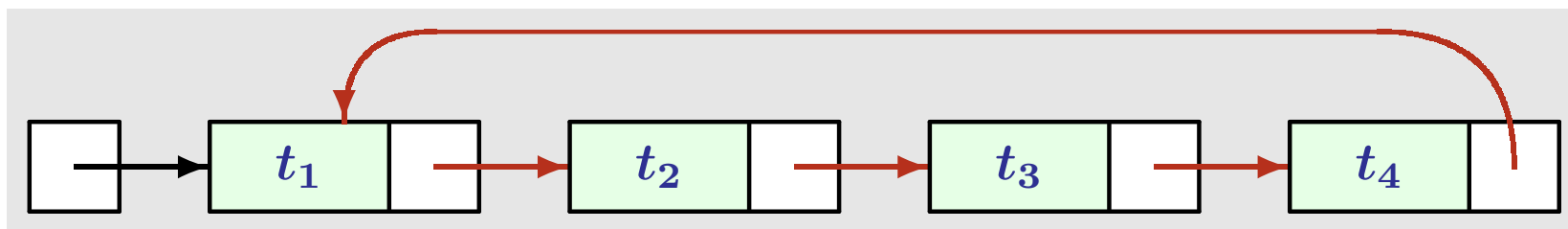
Zacyklenie listy

Przy konstruowaniu list przez pary należy **bardzo uważać na listy cykliczne!**

Lista jest realizowana przez wskaźniki:



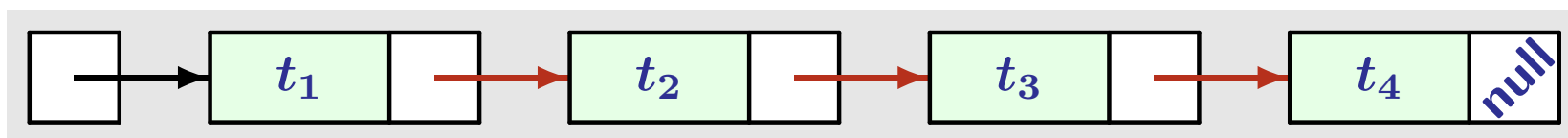
Źle ustawione wskaźniki mogą listę zacyklić:



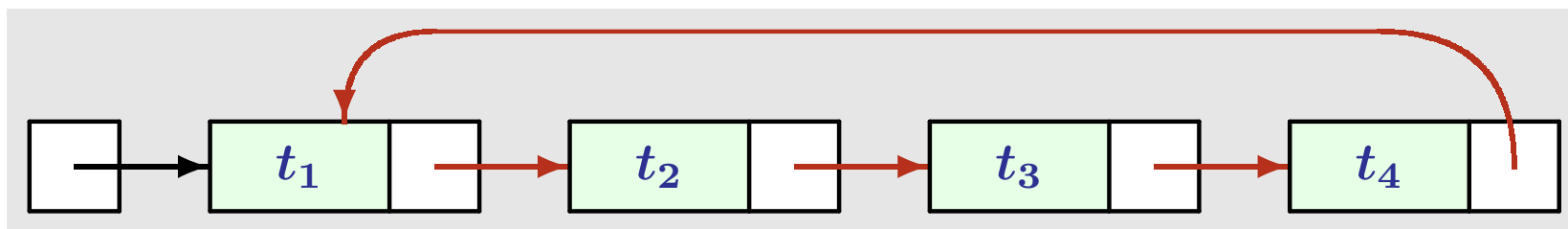
Zacyklenie listy

Przy konstruowaniu list przez pary należy **bardzo uważać na listy cykliczne!**

Lista jest realizowana przez wskaźniki:



Źle ustawione wskaźniki mogą listę zacyklić:

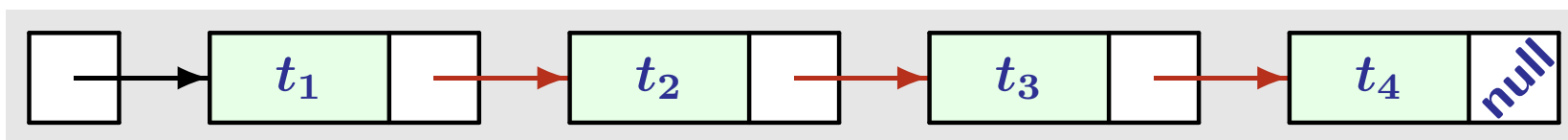


Wtedy dla obsługującego ją programu lista jest nieskończona.

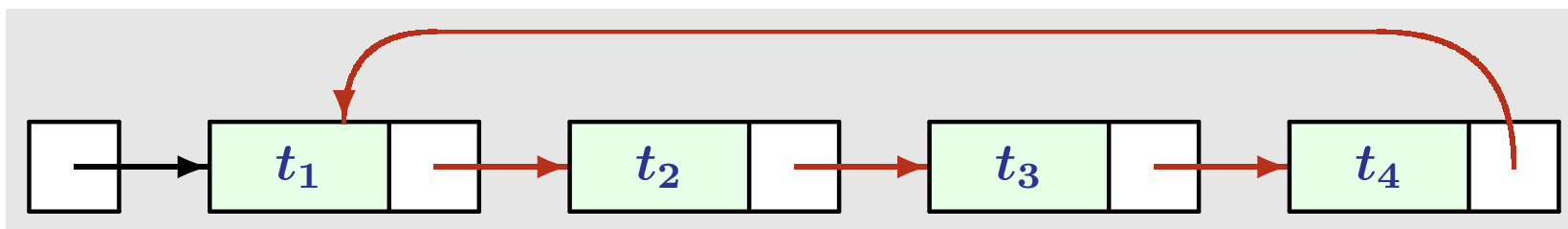
Zacyklenie listy

Przy konstruowaniu list przez pary należy **bardzo uważać na listy cykliczne!**

Lista jest realizowana przez wskaźniki:



Źle ustawione wskaźniki mogą listę zacyklić:



Wtedy dla obsługującego ją programu lista jest nieskończona.

Rozmnażanie funkcji po liście

Jak z funkcji działającej na pojedynczych elementach zrobić funkcję działającą na listach?

Rozmnażanie funkcji po liście

Jak z funkcji działającej na pojedynczych elementach zrobić funkcję działającą na listach?

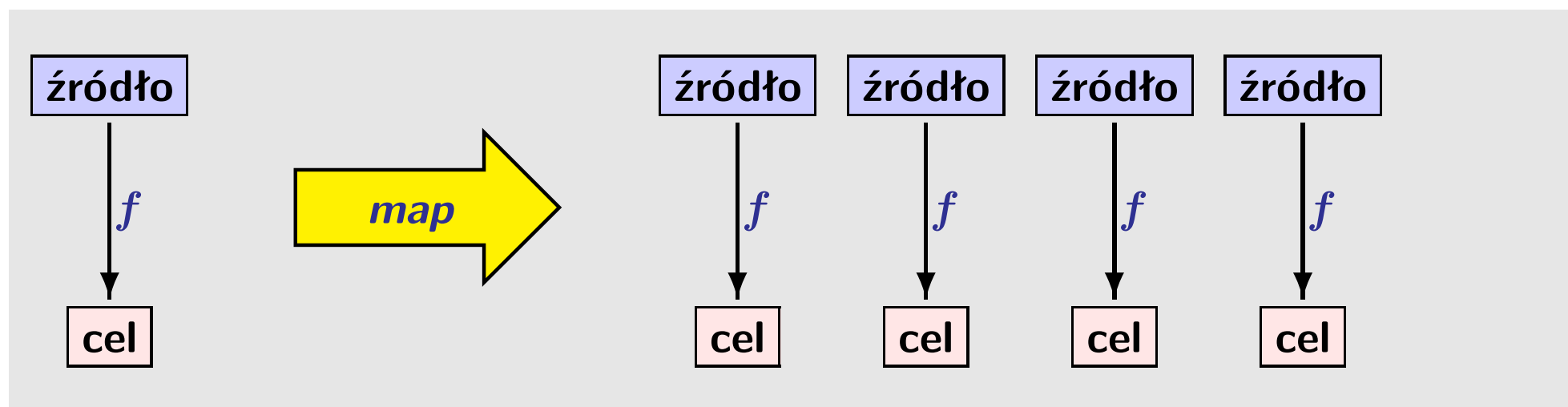
źródło

f

cel

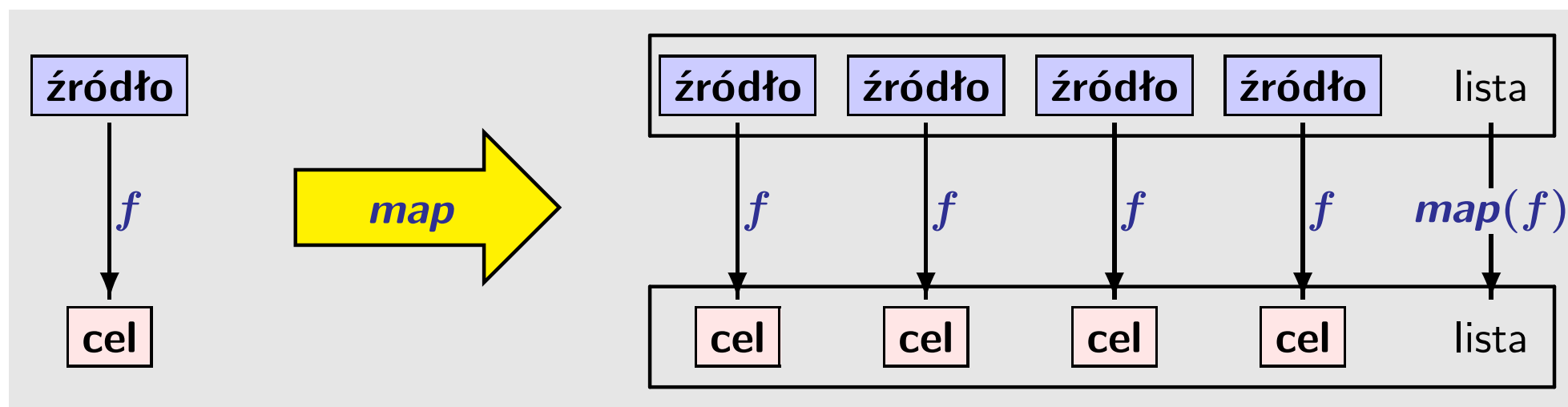
Rozmnażanie funkcji po liście

Jak z funkcji działającej na pojedynczych elementach zrobić funkcję działającą na listach?



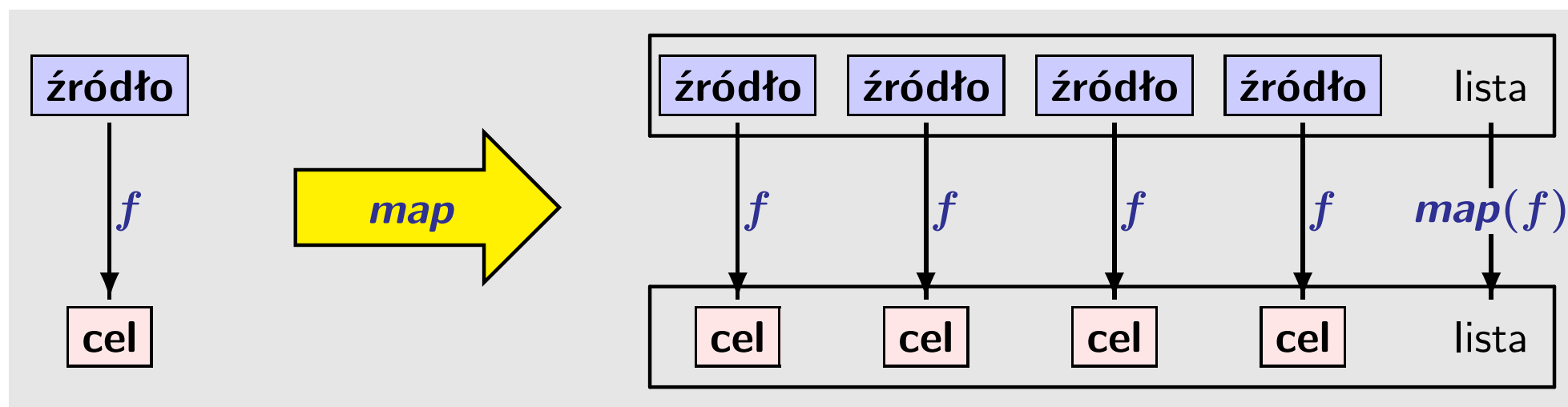
Rozmnażanie funkcji po liście

Jak z funkcji działającej na pojedynczych elementach zrobić funkcję działającą na listach?



Rozmnażanie funkcji po liście

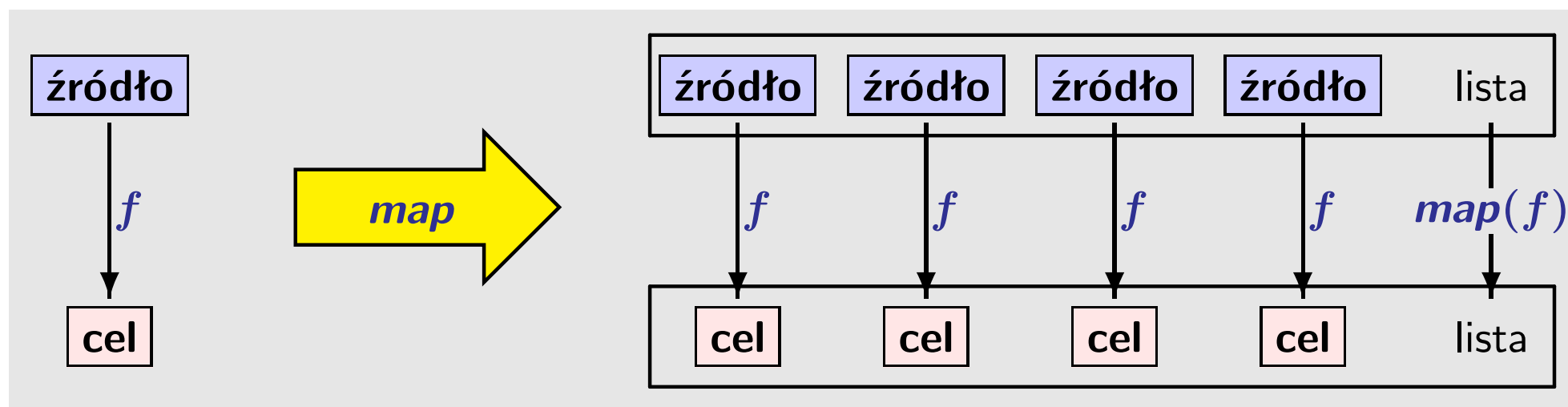
Jak z funkcji działającej na pojedynczych elementach zrobić funkcję działającą na listach?



$$\text{map}(f)[a_1, a_2, \dots, a_n] = [f(a_1), f(a_2), \dots, f(a_n)]$$

Rozmnażanie funkcji po liście

Jak z funkcji działającej na pojedynczych elementach zrobić funkcję działającą na listach?



$$map(f)[a_1, a_2, \dots, a_n] = [f(a_1), f(a_2), \dots, f(a_n)]$$

Mamy zdefiniować funkcję map działającą na funkcjach.

Na **dowolnej** funkcji i **dowolnej** liście.

Rozmnażanie funkcji po liście

$$\mathit{map}(f)[a_1, a_2, \dots, a_n] = [f(a_1), f(a_2), \dots, f(a_n)]$$

Na **dowolnej** funkcji i **dowolnej** liście.

Rozmnażanie funkcji po liście

$$\mathit{map}(f)[a_1, a_2, \dots, a_n] = [f(a_1), f(a_2), \dots, f(a_n)]$$

Na **dowolnej** funkcji i **dowolnej** liście.

Najprościej i najeleganciej — z użyciem interfejsu.

Rozmnażanie funkcji po liście

$$\mathit{map}(f)[a_1, a_2, \dots, a_n] = [f(a_1), f(a_2), \dots, f(a_n)]$$

Na **dowolnej** funkcji i **dowolnej** liście.

Najprościej i najeleganciej — z użyciem interfejsu. Jak w Jawie powiedzieć „dowolna funkcja”?

Rozmnażanie funkcji po liście

$$\text{map}(f)[a_1, a_2, \dots, a_n] = [f(a_1), f(a_2), \dots, f(a_n)]$$

Na **dowolnej** funkcji i **dowolnej** liście.

Najprościej i najeleganciej — z użyciem interfejsu. Jak w Jawie powiedzieć „dowolna funkcja”?

```
interface Fcja<S,T> { public T ff(S s); }
```

Rozmnażanie funkcji po liście

$$\mathit{map}(f)[a_1, a_2, \dots, a_n] = [f(a_1), f(a_2), \dots, f(a_n)]$$

Na **dowolnej** funkcji i **dowolnej** liście.

Najprościej i najeleganciej — z użyciem interfejsu. Jak w Jawie powiedzieć „dowolna funkcja”?

```
interface Fcja<S,T> { public T ff(S s); }
```

ŹRÓDŁO
argumenty funkcji

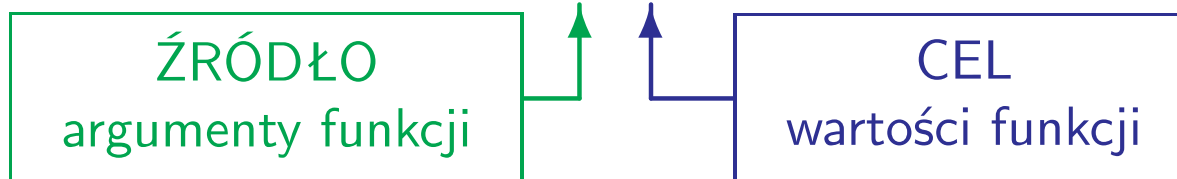
Rozmnażanie funkcji po liście

$$\text{map}(f)[a_1, a_2, \dots, a_n] = [f(a_1), f(a_2), \dots, f(a_n)]$$

Na **dowolnej** funkcji i **dowolnej** liście.

Najprościej i najeleganciej — z użyciem interfejsu. Jak w Jawie powiedzieć „dowolna funkcja”?

```
interface Fcja<S,T> { public T ff(S s); }
```



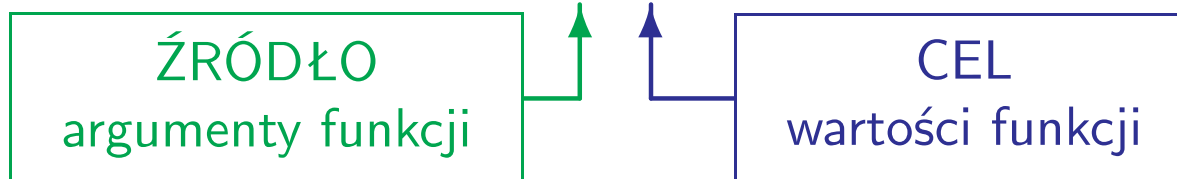
Rozmnażanie funkcji po liście

$$\text{map}(f)[a_1, a_2, \dots, a_n] = [f(a_1), f(a_2), \dots, f(a_n)]$$

Na **dowolnej** funkcji i **dowolnej** liście.

Najprościej i najeleganciej — z użyciem interfejsu. Jak w Jawie powiedzieć „dowolna funkcja”?

```
interface Fcja<S,T> { public T ff(S s); }
```



```
class Map<S,T> {
    public Fcja<ArrayList<S>, ArrayList<T>> map (Fcja<S,T> f) {
        . . . . .
    }
}
```

Rozmnażanie funkcji po liście

Definicja mapowania funkcji:

[illegible]

Rozmnażanie funkcji po liście

Definicja mapowania funkcji:

```
class Map<S,T> {  
    public Fcja<ArrayList<S>, ArrayList<T>> map (Fcja<S,T> f) {  
  
        Fcja<ArrayList<S>, ArrayList<T>> fList =  
            new Fcja<ArrayList<S>, ArrayList<T>>() {  
                public ArrayList<T> ff(ArrayList<S> sList) {  
                    var tList = new ArrayList<T>();  
                    for (S s : sList) tList.add(f.ff(s));  
                    return tList;  
                }  
            };  
        return fList;  
    }  
}
```

Rozmnażanie funkcji po liście

Definicja mapowania funkcji:

```
class Map<S,T> {
    public Fcja<ArrayList<S>, ArrayList<T>> map (Fcja<S,T> f) {
```

```
        Fcja<ArrayList<S>, ArrayList<T>> fList =
```

```
new Fcja<ArrayList<S>, ArrayList<T>>() {
    public ArrayList<T> ff(ArrayList<S> sList) {
        var tList = new ArrayList<T>();
        for (S s : sList) tList.add(f.ff(s));
        return tList;
    }
};
```

```
sList -> {
    var tList = new ArrayList<T>();
    for (S s : sList) tList.add(f.ff(s));
    return tList;
};
```

```
        return fList;
```

```
    }
}
```

Rozmnażanie funkcji po liście

Definicja mapowania funkcji:

```
class Map<S,T> {  
    public Fcja<ArrayList<S>, ArrayList<T>> map (Fcja<S,T> f) {  
  
        Fcja<ArrayList<S>, ArrayList<T>> fList =  
            sList -> {  
                var tList = new ArrayList<T>();  
                for (S s : sList) tList.add(f.ff(s));  
                return tList;  
            };  
        return fList;  
  
    }  
}
```

Rozmnażanie funkcji po liście

Definicja mapowania funkcji:

```
class Map<S,T> {
    public Fcja<ArrayList<S>, ArrayList<T>> map (Fcja<S,T> f) {

        Fcja<ArrayList<S>, ArrayList<T>> fList =
            sList -> {
                var tList = new ArrayList<T>();
                for (S s : sList) tList.add(f.ff(s));
                return tList;
            };
        return fList;
    }
}
```

Jeśli **X** jest interfejsem z **tylko jedną metodą f**, to konstrukcję

```
new X() { public f(...) {...coś tam...} }
```

można zastąpić przez wyrażenie Lambda

```
(...) -> {...coś tam...}
```

Rozmnażanie funkcji po liście

Przykład: lista napisów w listę ich długości:

```
new Map<String, Integer>().map(  
    napis -> napis.length()  
).ff
```

Rozmnażanie funkcji po liście

Przykład: lista napisów w listę ich długości:

```
new Map<String, Integer>().map(  
    napis -> napis.length()  
).ff
```

Przykład: lista liczb w listę ich zapisów ósemkowych:

```
new Map<Integer, String>().map(  
    liczba -> {  
        String licz = "";  
        while (liczba>0) {  
            licz = ""+(liczba%8)+licz;  
            liczba /= 8;  
        }  
        return licz;  
    }  
).ff
```

Rozmnażanie funkcji po liście

Przykład: lista napisów w listę ich długości:

```
new Map<String, Integer>().map(  
    napis -> napis.length()  
).ff
```

Przykład: lista liczb w listę ich zapisów ósemkowych:

```
new Map<Integer, String>().map(  
    liczba -> {  
        String licz = "";  
        while (liczba>0) {  
            licz = ""+(liczba%8)+licz;  
            liczba /= 8;  
        }  
        return licz;  
    }  
).ff
```

Maxmin uogólniony

Istnieją różne kryteria porównywania elementów różnych typów

Maxmin uogólniony

Istnieją różne kryteria porównywania elementów różnych typów; np.:

- **liczb** — ze względu na wielkość

Maxmin uogólniony

Istnieją różne kryteria porównywania elementów różnych typów; np.:

- **liczb** — ze względu na wielkość,
- **liczb** — ze względu na liczbę cyfr

Maxmin uogólniony

Istnieją różne kryteria porównywania elementów różnych typów; np.:

- **liczb** — ze względu na wielkość,
- **liczb** — ze względu na liczbę cyfr,
- **napisów** — alfabetycznie,
- itp.

Maxmin uogólniony

Istnieją różne kryteria porównywania elementów różnych typów; np.:

- **liczb** — ze względu na wielkość,
- **liczb** — ze względu na liczbę cyfr,
- **napisów** — alfabetycznie,
- itp.

Co one mają ze sobą wspólnego?

Maxmin uogólniony

Istnieją różne kryteria porównywania elementów różnych typów; np.:

- **liczb** — ze względu na wielkość,
- **liczb** — ze względu na liczbę cyfr,
- **napisów** — alfabetycznie,
- itp.

Co one mają ze sobą wspólnego?

Porównanie jest funkcją z **par** elementów **tego samego typu** do typu **logicznego**.

Maxmin uogólniony

Istnieją różne kryteria porównywania elementów różnych typów; np.:

- **liczb** — ze względu na wielkość,
- **liczb** — ze względu na liczbę cyfr,
- **napisów** — alfabetycznie,
- itp.

Co one mają ze sobą wspólnego?

Porównanie jest funkcją z **par** elementów **tego samego typu** do typu **logicznego**. Czyli należy do obiektu klasy implementującej taki interfejs:

```
interface Porownanie<Typ> {  
    public Boolean pred(Para<Typ,Typ> para);  
}
```

Maxmin uogólniony

Istnieją różne kryteria porównywania elementów różnych typów; np.:

- **liczb** — ze względu na wielkość,
- **liczb** — ze względu na liczbę cyfr,
- **napisów** — alfabetycznie,
- itp.

Co one mają ze sobą wspólnego?

Porównanie jest funkcją z **par** elementów **tego samego typu** do typu **logicznego**. Czyli należy do obiektu klasy implementującej taki interfejs:

```
interface Porownanie<Typ> {  
    public Boolean pred(Para<Typ,Typ> para);  
}
```

Jeśli umiemy porównywać elementy jakiegoś typu,
to umiemy znajdować maximum i minimum tablicy tego typu.

Maxmin uogólniony

```
class MaxMin<Typ> {  
    public Fcja<Typ[], Para<Typ,Typ>> maxmin (Porownanie<Typ> por) {  
        return tab -> {  
            var mm = new Para<Typ,Typ>(tab[0], tab[0]);  
            for (int i=1; i<tab.length; i++) {  
                if (por.pred(new Para<Typ,Typ>(mm.fst(),tab[i])))  
                    mm = new Para<Typ,Typ>(tab[i], mm.snd());  
                if (por.pred(new Para<Typ,Typ>(tab[i], mm.snd())))  
                    mm = new Para<Typ,Typ>(mm.fst(), tab[i]);  
            }  
            return mm;  
        };  
    }  
}
```

Maxmin uogólniony

```
class MaxMin<Typ> {  
    public Fcja<Typ[], Para<Typ,Typ>> maxmin (Porownanie<Typ> por) {  
        return tab -> {  
            var mm = new Para<Typ,Typ>(tab[0], tab[0]);  
            for (int i=1; i<tab.length; i++) {  
                if (por.pred(new Para<Typ,Typ>(mm.fst(), tab[i])))  
                    mm = new Para<Typ,Typ>(tab[i], mm.snd());  
                if (por.pred(new Para<Typ,Typ>(tab[i], mm.snd())))  
                    mm = new Para<Typ,Typ>(mm.fst(), tab[i]);  
            }  
            return mm;  
        };  
    }  
}
```

szukanie maximum



Maxmin uogólniony

```
class MaxMin<Typ> {  
    public Fcja<Typ[], Para<Typ,Typ>> maxmin (Porownanie<Typ> por) {  
        return tab -> {  
            var mm = new Para<Typ,Typ>(tab[0], tab[0]);  
            for (int i=1; i<tab.length; i++) {  
                if (por.pred(new Para<Typ,Typ>(mm.fst(), tab[i])))  
                    mm = new Para<Typ,Typ>(tab[i], mm.snd());  
                if (por.pred(new Para<Typ,Typ>(tab[i], mm.snd())))  
                    mm = new Para<Typ,Typ>(mm.fst(), tab[i]);  
            }  
            return mm;  
        };  
    }  
}
```

szukanie maximum

szukanie minimum

Maxmin uogólniony

```
class MaxMin<Typ> {  
    public Fcja<Typ[], Para<Typ,Typ>> maxmin (Porownanie<Typ> por) {  
        return tab -> {  
            var mm = new Para<Typ,Typ>(tab[0], tab[0]);  
            for (int i=1; i<tab.length; i++) {  
                if (por.pred(new Para<Typ,Typ>(mm.fst(), tab[i])))  
                    mm = new Para<Typ,Typ>(tab[i], mm.snd());  
                if (por.pred(new Para<Typ,Typ>(tab[i], mm.snd())))  
                    mm = new Para<Typ,Typ>(mm.fst(), tab[i]);  
            }  
            return mm;  
        };  
    }  
}
```

szukanie maximum

szukanie minimum

Sortowanie uogólnione

Zadanie samodzielne:

Zaprogramować **sortowanie** tablicy działające dla dowolnego typu elementów i dowolnego porównania.

Przetestować co najmniej dla liczb oraz dla napisów (ale oczywiście powinno działać również dla innych typów i porównań).

Projekt programistyczny

Projekt programistyczny

oczywiste prawdy	wnioski
programowanie ma sens dla przetwarzania dużych zbiorów danych	

Projekt programistyczny

oczywiste prawdy	wnioski
programowanie ma sens dla przetwarzania dużych zbiorów danych	programowania nie można się uczyć wyłącznie na minidanych w trybie klawiatura → ekran

Projekt programistyczny

oczywiste prawdy	wnioski
programowanie ma sens dla przetwarzania dużych zbiorów danych	programowania nie można się uczyć wyłącznie na minidanych w trybie klawiatura → ekran
na małych danych wszystko śmiga	

Projekt programistyczny

oczywiste prawdy	wnioski
programowanie ma sens dla przetwarzania dużych zbiorów danych	programowania nie można się uczyć wyłącznie na minidanych w trybie klawiatura → ekran
na małych danych wszystko śmiga — dopiero na dużych można porównać jakość różnych rozwiązań	

Projekt programistyczny

oczywiste prawdy	wnioski
programowanie ma sens dla przetwarzania dużych zbiorów danych	programowania nie można się uczyć wyłącznie na minidanych w trybie klawiatura → ekran
na małych danych wszystko śmiga — dopiero na dużych można porównać jakość różnych rozwiązań	dużych danych i wyników nie da się objąć „rzutem oka”

Projekt programistyczny

oczywiste prawdy	wnioski
programowanie ma sens dla przetwarzania dużych zbiorów danych	programowania nie można się uczyć wyłącznie na minidanych w trybie klawiatura → ekran
na małych danych wszystko śmiga — dopiero na dużych można porównać jakość różnych rozwiązań	dużych danych i wyników nie da się objąć „rzutem oka” — potrzebne są pomocnicze programy dla oceny

Projekt programistyczny

oczywiste prawdy	wnioski
programowanie ma sens dla przetwarzania dużych zbiorów danych	programowania nie można się uczyć wyłącznie na minidanych w trybie klawiatura → ekran
na małych danych wszystko śmiga — dopiero na dużych można porównać jakość różnych rozwiązań	dużych danych i wyników nie da się objąć „rzutem oka” — potrzebne są pomocnicze programy dla oceny
program jest cenny, jeśli daje się wielokrotnie zastosować do różnych danych (<i>reusable</i>)	

Projekt programistyczny

oczywiste prawdy	wnioski
programowanie ma sens dla przetwarzania dużych zbiorów danych	programowania nie można się uczyć wyłącznie na minidanych w trybie klawiatura → ekran
na małych danych wszystko śmiga — dopiero na dużych można porównać jakość różnych rozwiązań	dużych danych i wyników nie da się objąć „rzutem oka” — potrzebne są pomocnicze programy dla oceny
program jest cenny, jeśli daje się wielokrotnie zastosować do różnych danych (<i>reusable</i>)	program powinien być tak napisany, żeby jego zastosowanie do innych okoliczności wymagało wymiany tylko niewielkiego fragmentu (<i>modularyzacja</i>)

Projekt programistyczny

Narzędzia modularyzacji

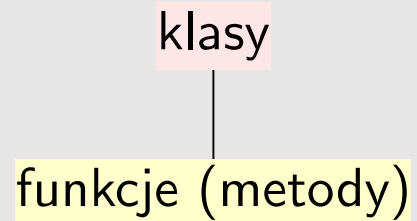
Projekt programistyczny

Narzędzia modularyzacji

funkcje (metody)

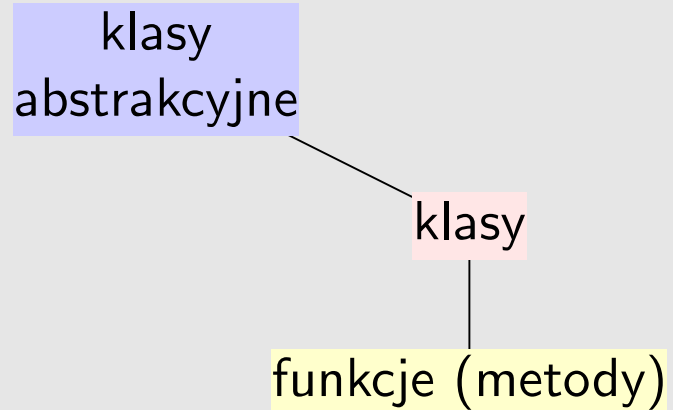
Projekt programistyczny

Narzędzia modularyzacji



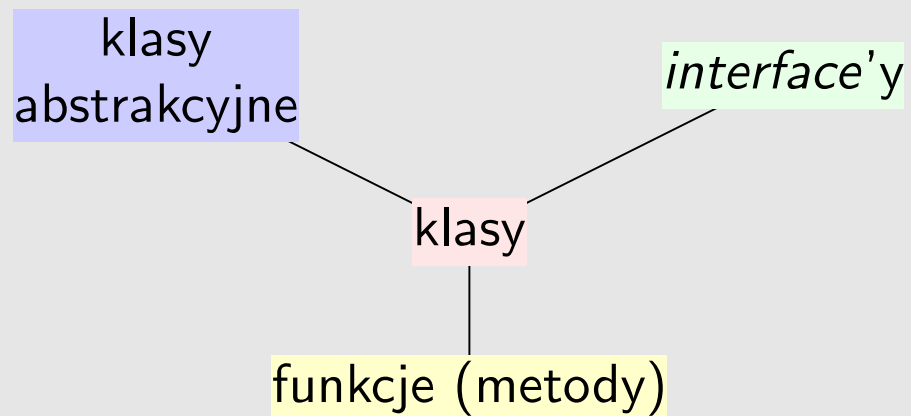
Projekt programistyczny

Narzędzia modularyzacji



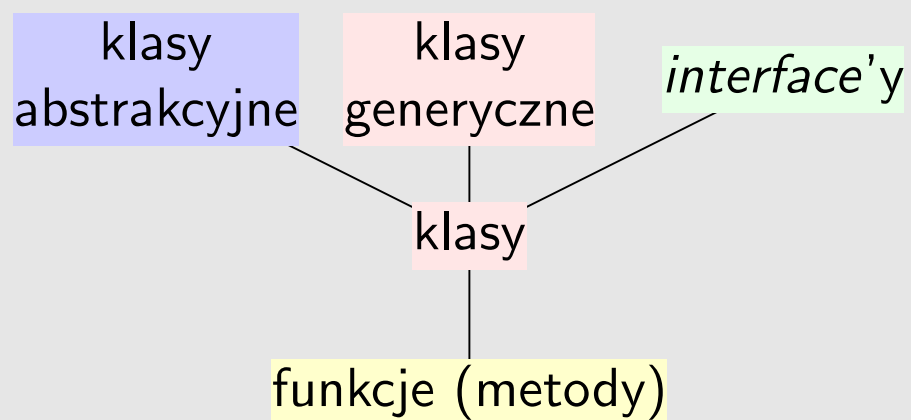
Projekt programistyczny

Narzędzia modularyzacji



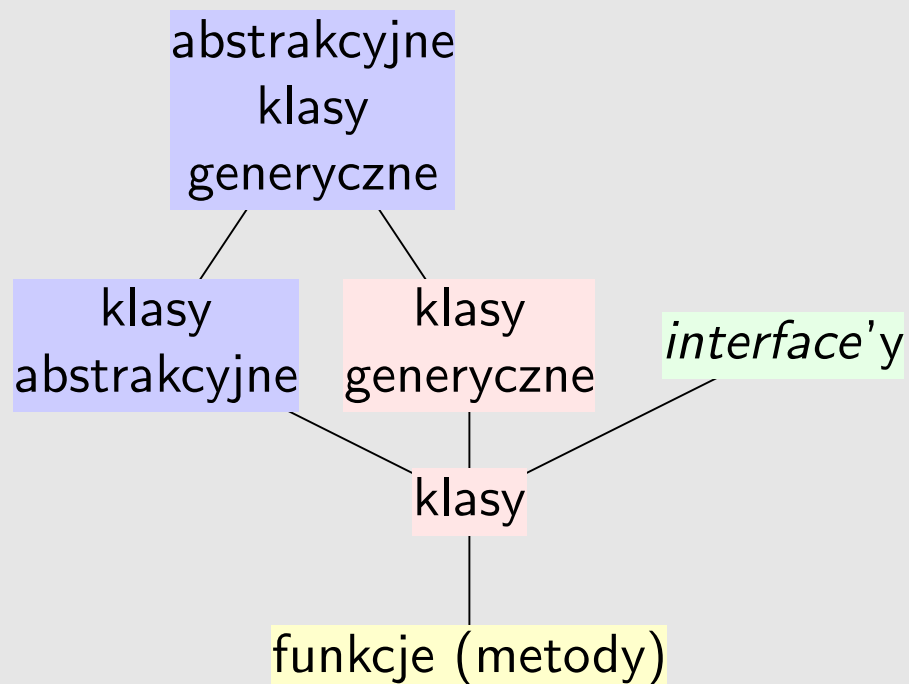
Projekt programistyczny

Narzędzia modularyzacji



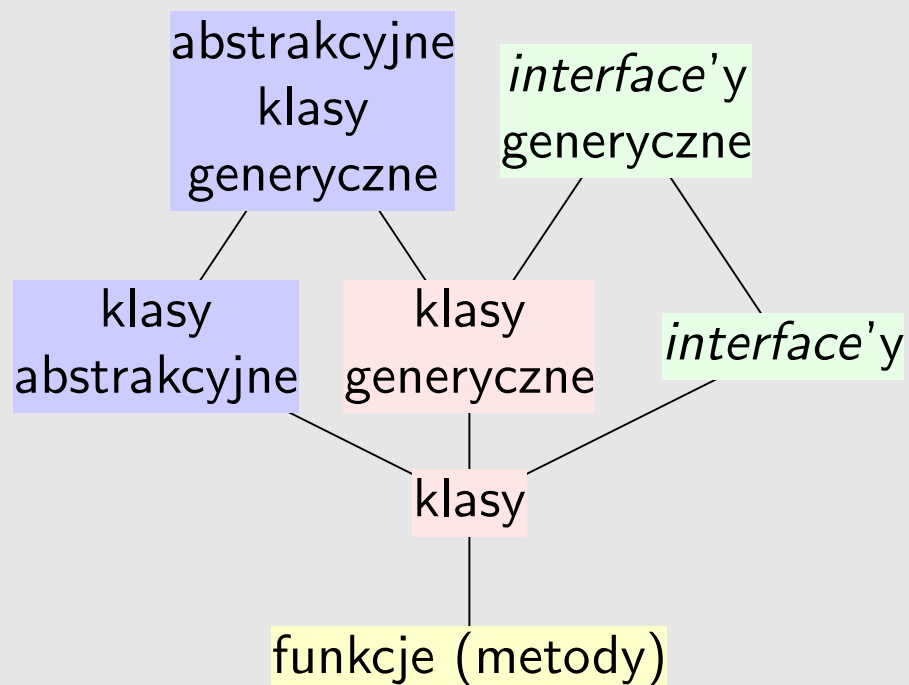
Projekt programistyczny

Narzędzia modularyzacji



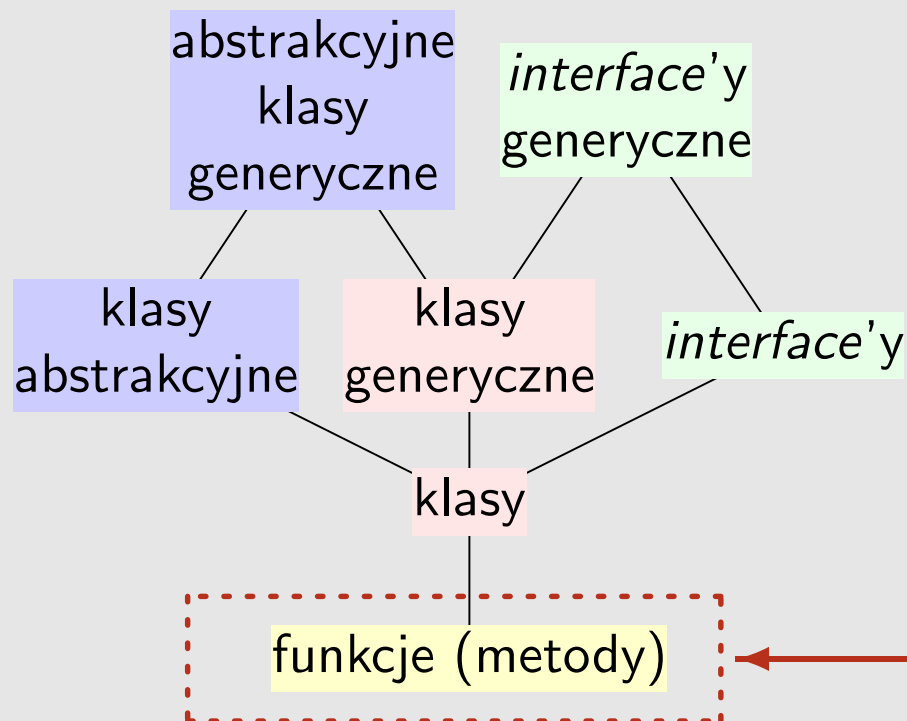
Projekt programistyczny

Narzędzia modularyzacji



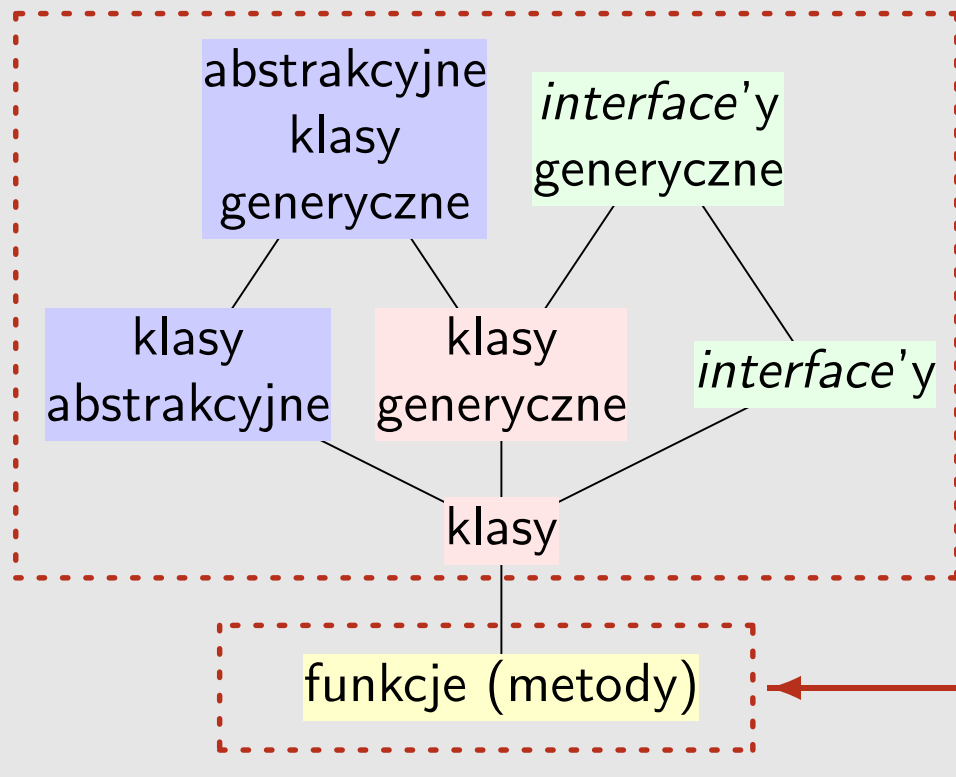
Projekt programistyczny

Narzędzia modularyzacji



Projekt programistyczny

Narzędzia modularyzacji



dodatkowo w Jawie i
w innych obiektowych

dostępne w C i w innych
językach strukturalnych