Procedury
i
metody

Tomasz Borzyszkowski

Typy obiektowe

W Java prócz typów prostych możemy tworzyć typy „obiektowe”, tzw. klasy.
Konstrukcja klasy wygląda tak:

```
class NazwaKlasy {
   typ pole_1;
   ...
   typ pole_n;
   typ metoda_1(typ par11, ..., typ par_i1){ ... }
   ...
   typ metoda_k(typ par1k, ..., typ par_ik){ ... }
}
```

Krotka z obsługą.

Zobacz: **NowaKlasa.java**

Rola procedur w programowaniu

Rola procedur (funkcji):
- pomagają podzielić kod programu na mniejsze, logicznie wydzielone zadania
- ukrywają szczegóły pewnych operacji tam, gdzie ich znajomość nie jest niezbędna
- zwiększają przejrzystość kodu programu i ułatwiają wprowadzanie w nim zmian

Procedury – podstawowe cechy:
- posiada unikatową sygnaturę
- składa się z ciągu instrukcji implementującego dobrze określoną operację/zadanie
- może być zależna od argumentów i zwracać wynik

Zobacz: **Sygnatury.java**

Sygnatury i przeciążanie

W różnych językach programowania procedury są rozpoznawane po różnych cechach. Zespół cech powalających dwie procedury w języku programowania nazywamy **sygnaturą**.

W Java sygnatura składa się z:
- nazwy procedury
- typy parametrów wejściowych

W C i C++ tylko nazwa procedury.

Przez analogię do C++ często procedury o tych samych nazwach w tej samej przestrzeni nazw nazywany przeciążonymi.

Zobacz: **Sygnatury2.java**

Przekazywanie parametrów

Obiekty jako parametry metod. Ponieważ klasy w Java są traktowane jak typy w tradycyjnych językach programowania, to można przekazywać obiekty (parametry typu klasowego) do metod i zwracać obiekty jako wartości. **Zobacz**: **ParameterDemo.java**

Przekazywanie parametrów metod przez wartość i zmienną. W Java sposób przekazywania parametrów do metody zależy od ich typu. Parametry typów prostych zawsze są przekazywane przez wartość, natomiast przekazywanie obiektów do metod zawsze odbywa się przez zmienną.

Przekazywanie przez wartość: przekazywane wartości nie ulegają trwałym zmianom - *po wywołaniu metody wartość jak przed*

Przekazywanie przez zmienną: przekazywane wartości ulegają trwałym zmianom - *po wywołaniu metody wartość się zmienia*

Zobacz: **CallByDemo.java**

Opakowanie typów prostych

W językach obiektowych (Java, C#, ...) by przekazać zmienną typu prostego do metody przez zmienną stosujemy metodę: **boxing-unboxing**.

W Java dla każdego typu prostego istnieje typ obiektowy opakowujący dany typ prosty. Np.:

int	Integer
double	Double
float	Float

Zobacz: **Boxing.java**

Znajdź więcej przykładów.

Co to jest REKURSJA ?

Nieformalnie:

Rekursja jest metodą algorytmicznego definiowania funkcji, w której algorytm odwołuje się, bezpośrednio lub pośrednio do samego siebie.

Przykład: Funkcja silnia **n!**

silnia(0) = 1
silnia(n) = n * silnia(n – 1), dla n > 0

Najpierw obliczamy tę samą funkcję z mniejszym/łatwiejszym argumentem, następnie wynik używamy do policzenia funkcji z trudniejszym argumentem.

Sprowadzamy obliczenia do prostrzego przypadku ! ! !

Zobacz: **Silnia.java**

Silnia: Obliczenia

silnia(5) = 5 * silnia(4)
= 5 * (4 * silnia(3))
= 5 * (4 * (3 * silnia(2))
= 5 * (4 * (3 * (2 * silnia(1)))
= 5 * (4 * (3 * (2 * (1 * silnia(0))))
= 5 * (4 * (3 * (2 * (1 * 1))))
= 5 * (4 * (3 * (2 * 1)))
= 5 * (4 * (3 * 2))
= 5 * (4 * 6)
= 5 * 24
= 120

Zła definicja

Niech:

$f(0) = 3$

$f(n) = 3 * f(n - 2), \quad \text{dla } n > 0$

Wówczas:

$f(5) = 3 * f(3)$

$= 3 * (3 * f(1))$

$= 3 * (3 * (3 * f(-1)))$

$= 3 * (3 * (3 * (3 * f(-3))))$

...

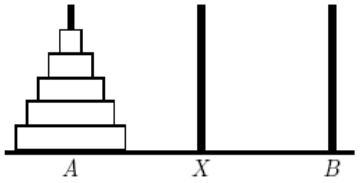
Ważne by definicja po pewnej skończonej liczbie kroków sprowadzała się do tzw. przypadku bazowego – tj. obliczenia, które już nie potrzebuje obliczania wartości f na innym argumencie

Tu potrzebna wartość dla f(1)

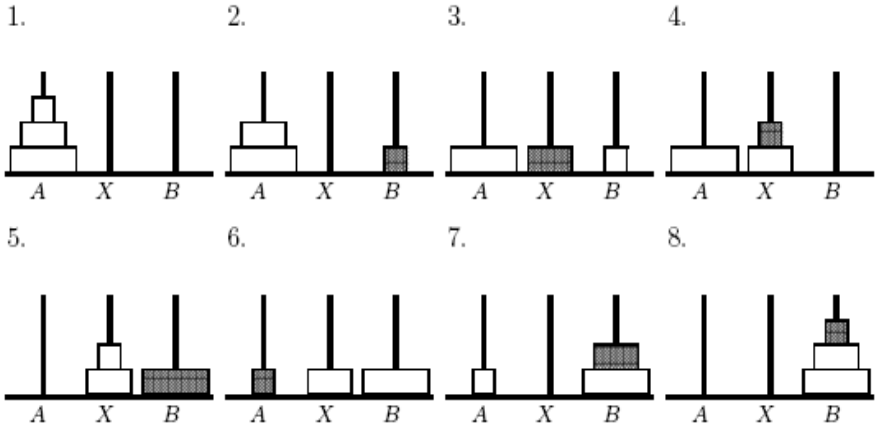
Wieże Hanoi

Problem: (Edouard Lucas, 1883)

Dany jest stos zbudowany z n krążków (z dziurką w środku) o coraz mniejszej średnicy nawleczonych na jeden z trzech patyków.
Zadanie polega na przeniesieniu stosu na inny patyk przestrzegając następujących reguł:
w jednym kroku wolno przenieść tylko jeden krążek
nigdy nie wolno kłaść większego krążka na mniejszy



Wieże Hanoi Rozwiązanie dla n = 3



Wieże Hanoi Pomysł

Pomysł:

Gdy „zapomni się" na początku o najniższym (największym) krążku wieży, to zostaje do przeniesienia wieża o wysokości n – 1

Postępowanie:

przenieś wieżę wysokości n – 1 z patyka A na patyk X przy pomocy patyka B

przenieś pozostały (największy) krążek z A na B

przenieś wieżę wysokości n – 1 z patyka X na patyk B przy pomocy patyka A

Wieże Hanoi Algorytm

Algorytm: Hanoi
wejście: n ∈ Nat, patyki A, B, X.
wyjście: kolejność ruchów

```
Proc Hanoi(n : Nat, A, B, X : Patyk)
begin
   if n > 0 then begin
      Hanoi(n-1,A,X,B);
      write „Krążek z A na B";
      Hanoi(n-1,X,B,A);
   end
end
```

Algorytm nie dostarcza wyniku, ale podaje kolejność ruchów rozwiązujących problem wież Hanoi.
Zobacz realizację w Java: **Hanoi.java**

Algorytm Euklidesa Koncepcja

Dane są dwie liczby naturalne a i b. Do obliczenia jest największy wspólny dzielnik liczb a i b: NWD(a, b).

NWD(a, 0) = a
NWD(a, b) = NWD(b, a mod b), dla a > b

Czy to jest dobrze zdefiniowane?

Dotychczas: OK. ponieważ n – 1 < n, to kiedyś dostaniemy argument, który skończy rekursję.
Teraz: dwa argumenty; jeżeli rekursja zostanie wywołana, drugi argument, b, staje się pierwszym argumentem.
Ale: **b + (a mod b) < a + b, dla a > b**

tzn. kiedyś drugi argument musi być równy 0 i rekursja się zakończy.

Algorytm Euklidesa Koncepcja

Dane są dwie liczby naturalne a i b. Do obliczenia jest największy wspólny dzielnik liczb a i b: NWD(a, b).

NWD(a, 0) = a
NWD(a, b) = NWD(b, a mod b), dla a > b

Czy to jest dobrze zdefiniowane?

Dotychczas: OK. ponieważ n – 1 < n, to kiedyś dostaniemy argument, który skończy rekursję.
Teraz: dwa argumenty; jeżeli rekursja zostanie wywołana, drugi argument, b, staje się pierwszym argumentem.
Ale: **b + (a mod b) < a + b, dla a > b**

tzn. kiedyś drugi argument musi być równy 0 i rekursja się zakończy.

Algorytm Euklidesa Obliczenia

NWD(206, 40) = NWD(40, 6)
= NWD(6, 4)
= NWD(4, 2)
= NWD(2, 0)
= **2**

NWD(40, 206) = NWD(206, 40)
=
...

Algorytm Euklidesa Algorytm

Algorytm: Euklidesa
wejście: a > b ϵ Nat
wyjście: NWD(a, b).

```
Proc Euklides(a, b : Nat) : Nat
begin
   if b = 0 then a else Euklides(b, a mod b)
end
```

Ilość wykonań pętli: logarytmiczna z argumentem **max(a, b)**.
ale: założenie, że operacja dzielenia „kosztuje" tylko jeden krok, jest nierealistyczne. Rozważając problem dzielenia w zależności od wielkości liczb, otrzymujemy:

$$T_{Euklid} \in O(n^2)$$

Patrz: **Euklides.java**

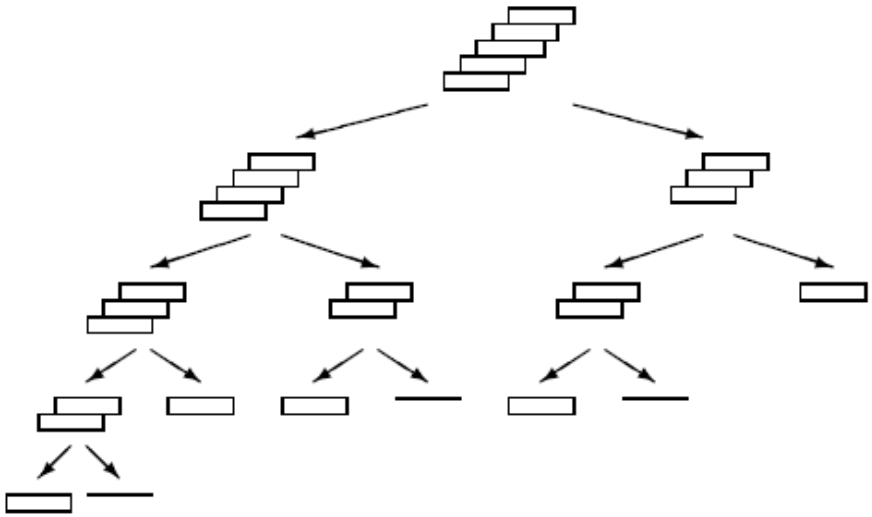
Ciąg Fibonacciego

Problem:
Dane są schody z **n** stopniami. Przy każdym kroku można wybrać, czy wejść jeden, czy dwa stopnie naraz.
Ile jest możliwości **M(n)** wejścia po schodach o **n** stopniach?

Rozwiązanie rekurencyjne:
W pierwszym kroku wchodzi się albo
jeden stopień:
w tym wypadku pozostają do wejścia schody z n – 1 stopniami, albo
dwa stopnie:
wtedy pozostaną do wejścia schody z n – 2 stopniami

W sumie mamy:
M(n – 1) + M(n – 2) możliwości.

Ciąg Fibonacciego n = 5



Ciąg Fibonacciego Algorytm

Algorytm: Fib
wejście: n ϵ Nat
wyjście: Fib(n).

```
Proc Fib(n : Nat) : Nat
begin
   if n < 2 then n  else Fib(n – 1) + Fib(n – 2)
end
```

Korzystaliśmy z właśności:

Fib(0) = 0
Fib(1) = 1
Fib(n) = Fib(n – 1) + Fib(n – 2)

Ciąg Fibonacciego szybciej

Pomysł:

f0 = Fib(n - 2), f1 = Fib(n – 1), to Fib(n) = f0 + f1

Algorytm: Fib2

wejście: n ϵ Nat

wyjście: Fib(n)

```
Proc Fib2(n : Nat) : Nat
begin
   f0 := 0; f1 := 1;
     for i = 2, . . ., n
     begin
        w := f0 + f1;
        f0 := f1;
        f1 := w;
     end
     return w;
end
```

$T_{Fib2} \in O(n)$

Patrz: **Fibonacci.java**

Ciąg Fibonacciego jeszcze szybciej

Obserwacja:

$$\begin{bmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_{n+3} & F_{n+2} \\ F_{n+2} & F_{n+1} \end{bmatrix} \quad \text{oraz} \quad \begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

to $\begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$ dla n > 0

Koszt:

$T_{Fib3} \in O(\log_2 n)$

Zadanie: napisać algorytm korzystający z pow. obserwacji

Funkcja Ackermanna

Definicja

$A(0, n) = n + 1$

$A(m, 0) = A(m - 1, 1)$

$A(m, n) = A(m - 1, A(m, n - 1))$

Patrz: **Ackermann.java**

Tak zdefiniowana funkcja posiada wartość dla każdej pary liczb naturalnych.
Funkcja ta niezwykle szybko rośnie:

m/n	0	1	2	3	4	n
0	1	2	3	4	5	n + 1
1	2	3	4	5	6	n + 2
2	3	5	7	9	11	2n + 3
3	5	13	29	61	125	$2^{n+3}-3$
4	13	65533	$2^{65536}-3$	$2^{2^{65536}}-3$	A(3, A(4, 3))	$\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{n+3}-3$