

# **Groovy: funkcje i metody**

© Krzysztof Barteczko, PJWSTK 2012 – 2017

# Funkcje

Funkcje definiujemy za pomocą składni:

```
(def | Typ) funcName (parameters) { // nagłówek  
  // ciało funkcji = instrukcje składające się na funkcję  
}
```

i wywołujemy:

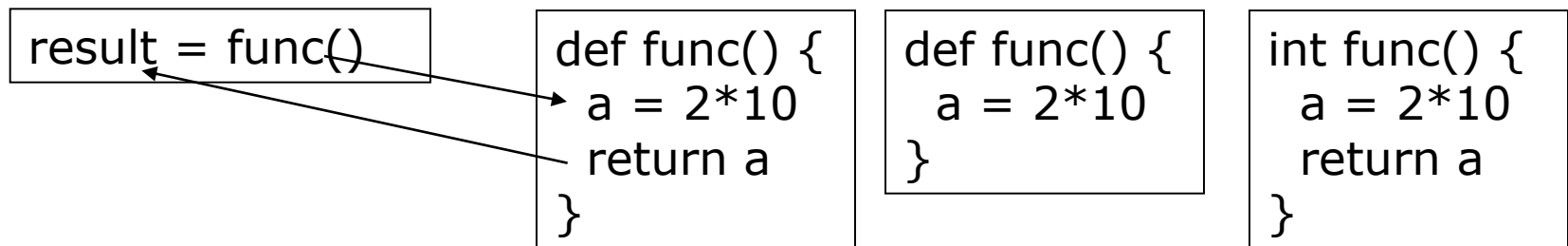
***funcName(arguments)***

W programowaniu obiektowym funkcje nazywamy metodami.

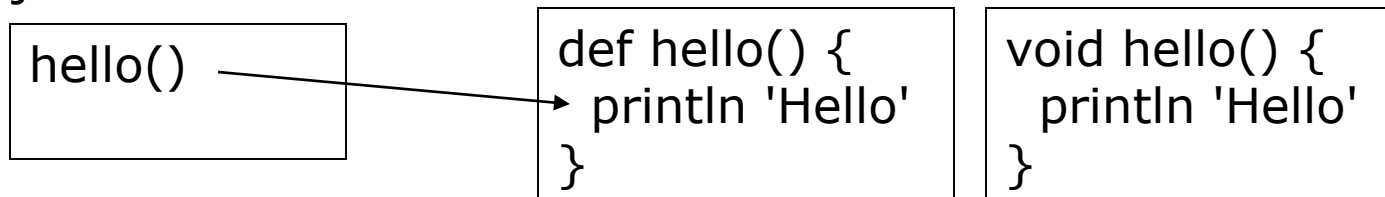
W odniesieniu do definicji zawartych w prostych skryptach (bez definicji klas) będziemy stosować nazwę *funkcja*.

# Wywołanie i wynik funkcji

Wywołanie funkcji polega na przekazaniu sterowania do kodu ciała funkcji. Na skutek wykonania tego kodu może powstać jakaś wartość i może ona być zwrócona do miejsca wywołania.



Funkcja może nie mieć żadnego wyniku. Wtedy jej typem wyniku jest Void.



Użycie def w miejsce typu wyniku oznacza dowolny typ.

Wykonanie funkcji kończy się i sterowanie jest zwracane w miejsce wywołania gdy zakończy się ciało funkcji lub wykonana zostanie instrukcja return.

Wynikiem funkcji jest wartość ostatniego wyrażenia wykonanego w ciele funkcji lub wartość wyrażenia podanego w instrukcji return.

# Parametry

Definiując funkcję (metodę) podajemy w nawiasach listę jej parametrów, przy czym można pominąć typy:

```
def func(a, b, c) {  
    // ...  
}
```

Np. wystarczająca definicja metody main w klasie może wyglądać tak:

```
static main(a) { // pominięte typy wyniku i argumentu  
    // ...  
}
```

Oczywiście, można też typy podawać, co zwiększa możliwość kontroli w fazie kompilacji.

# Argumenty

Jeżeli funkcja (metoda) ma parametry, to w jej wywołaniu można pominąć nawiasy okrągłe (jeśli składnia jest jednoznaczna)

`println 'Groovy' ⇔ println('Groovy')`

Wywołanie metod `getNnn()` i `setNnn(arg)` z dowolnej klasy (również stworzonej w Javie) może być zastąpione bezpośrednim operowaniem na właściwościach:

`s = f.getText() ⇔ s = f.text`

`f.text = 'Ala ma kota'`

# Wołanie metod jako operowanie na właściwościach - przykład 1

```
import javax.swing.*
```

```
def b = new JButton('AAA') // klasa ma metody get/setText
```

```
def l = new JLabel('BBB') // klasa ma metody get/setText
```

```
class A {  
    private String text = 'initial'  
    String getText() { return text }  
    void setText(String s) { text = s }  
}
```

```
def a = new A()  
for (c in [b, l, a]) println c.text  
for (c in [b, l, a]) c.text = "change $c.text"  
for (c in [b, l, a]) println c.text
```



```
AAA  
BBB  
initial  
change AAA  
change BBB  
change initial
```

# Wołanie metod jako operowanie na właściwościach - przykład 2

Szczególnie użyteczne, gdy mamy długie nazwy metod

// Copy – Paste to/from clipboard

```
import java.awt.datatransfer.*
import java.awt.Toolkit

def setClipboardText(txt) {
  Toolkit.getDefaultToolkit().getSystemClipboard()
    .setContents(new StringSelection(txt), null);
}

def getClipboardText() {
  Toolkit.getDefaultToolkit().systemClipboard.getContents(null)
    .getTransferData(DataFlavor.stringFlavor)
}

// zasada get-set dotyczy też wołanie metod ze skryptu
clipboardText = 'Ala ma kota'
println clipboardText

// Wynik na konsoli:
Ala ma kota
```

# Dynamic dispatch

A gdyby napisać tak:

```
import java.awt.Toolkit
Toolkit.clipboard.text = 'Ala ma kota'
println Toolkit.clipboard.text
```

to nie byłoby błędu w kompilacji, ale w fazie wykonania pojawiłby się błąd. W klasie Toolkit nie ma metody getClipboard().

O ile nie użyto adnotacji, wymuszających statyczną kontrolę typów w fazie kompilacji, Groovy rozwiązuje wszystkie wywołania metod/funkcji w fazie wykonania i dynamicznie dobiera odpowiednie metody (jeśli są), a gdy ich nie ma pojawia się wyjątek `MissingMethodException`.



## Dynamic dispatch (2)

Dynamiczność jest zrozumiała, bowiem w Groovy w trakcie wykonania do klas można dodawać nowe metody. Więcej o tym w wykładzie o metaprogramowaniu, teraz zobaczmy, że jednak możemy z powodzeniem wykonać fragment z poprzedniego slajdu, jeżeli tylko do odpowiednich klas dodamy odpowiednie metody:

```
import java.awt.datatransfer.*
import java.awt.Toolkit

// Dodanie statycznej metody getClipboard() do Toolkit
// metoda zwraca referencję typu Clipboard
Toolkit.metaClass.static.getClipboard = {
    Toolkit.getDefaultToolkit().getSystemClipboard()
}

// dodanie metod getText i setText do klasy Clipboard
Clipboard.metaClass.getText = {
    getContents(null).getTransferData(DataFlavor.stringFlavor)
}

Clipboard.metaClass.setText = { txt->
    getContents(null).setContents(new StringSelection(txt), null)
}
// i teraz bez problemu działa:
Toolkit.clipboard.text = 'Ala ma kota'
println Toolkit.clipboard.text
```

# Multiple dispatch

W Javie wywołanie metod z klas jest dobierane na podstawie aktualnego typu obiektu, na rzecz którego metoda jest wołana (polimorfizm). Jeśli chodzi o argumenty, to Java bazuje na ich formalnym (statycznym) typie. Nie ma polimorfizmu "po argumentach".

W Groovy metody są dopierane dynamicznie, bazując zarówno na aktualnym typie obiektu na rzecz którego metoda jest wołana, jak i na aktualnych (runtime) typach argumentów. Nazywa się to "multiple dispatch" albo "multimethods".

[Prosty przykład multimethods](#)

[Dla zainteresowanych – większy przykład \(na tle Javy\)](#)

# Funkcje ze zmienną liczbą argumentów

Definicja:

`def fun(... args) / def fun(def ... args) / String fun(int ... args)`

Przykład - definicja, wywołanie i dostęp do argumentów:

```
vaf(1,2,5)
vaf('Dog', 10, 'Cat', 5)

def vaf( ... args) {
  s = args.size()
  println "Args num: $s"
  println 'First   : ' + args[0]
  println 'Last    : ' + args[s-1]
  for (a in args) {
    print a + ' '
  }
  println()
}
```

Args num: 3  
First : 1  
Last : 5  
1 2 5  
Args num: 4  
First : Dog  
Last : 5  
Dog 10 Cat 5

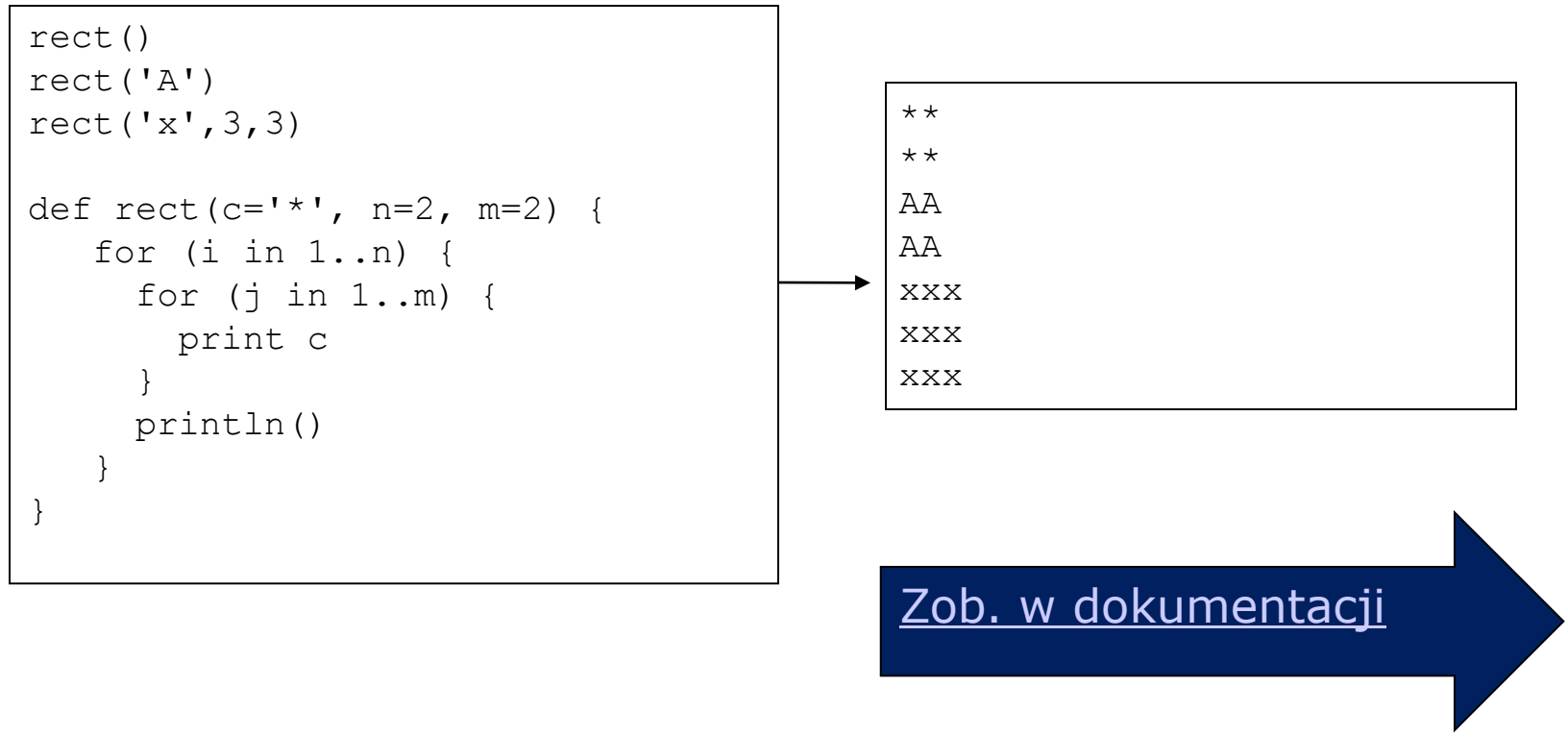
[Zob. w dokumentacji](#)

# Argumenty domyślne

Przy definicji funkcji jednemu lub kilku ostatnim (albo i wszystkim) argumentom można nadac wartości domyślne.

```
rect()
rect('A')
rect('x',3,3)

def rect(c='*', n=2, m=2) {
  for (i in 1..n) {
    for (j in 1..m) {
      print c
    }
    println()
  }
}
```



```
**
**
AA
AA
xxx
xxx
xxx
```

[Zob. w dokumentacji](#)

# Nazwane parametry

Funkcję z parametrem typu Map można wywoływać podając nazwane parametry:


id1 : val1, id2 : val2, ...

Wygoda:

nie trzeba podawać wszystkich, nie trzeba pamiętać pozycji

```
rect char: 'o'
rect cols: 5, rows: 1
rect cols: 7, char: 'X'

def rect(amap) {
  if (!amap) println 'Invalid arg'
  def c = amap.char ? : '*'
  def n = amap.rows ? : 2
  def m = amap.cols ? : 2
  for (i in 1..n) {
    for (j in 1..m) print c
    println()
  }
}
```



```
oo
oo
*****
XXXXXXXXX
XXXXXXXXX
```

# Wiązania skryptu

Zmienne wprowadzane w skrypcie i jego funkcjach bez nazwy typu (w tym bez def) są dodawane do wiązań skryptu i dostępne od momentu utworzenia również w innych funkcjach skryptu.

```
a = 1
b = 2                // a i b dodane do wiązań (bindings)
func1()
println "$a $b"

def func1() {
  println "$a $b"    // te same a i b - pobrane z bindings
  a = 10; b = 11;
}
```

Wynik:

```
1 2
10 11
```

Bindings zapewniają również komunikację pomiędzy różnymi skryptami, np. ze skryptami uruchamianymi dynamicznie w trakcie wykonania programu.

# Zmienne lokalne

Zmienne wprowadzane z nazwą typu lub słowem `def` są zmiennymi lokalnymi, widocznymi tylko w danym bloku (i blokach w nim zawartych).

```
def c = 100
def d = 111
func2()
println "$c $d"
```

```
def func2() {
  // println "$c $d"
  def c = 77, d = 88
  println "$c $d"
}
```

To są różne bloki!



```
// Błąd wykonania: nieznane zmienne c i d
// Nowe, lokalne dla tego bloku c i d
```

Wynik:

```
77 88
100 111
```

W bardziej skomplikowanych skryptach warto używać `def`, aby przypadkowo nie popsuć wartości jakichś zmiennych

# Adnotacja @Field

Użycie adnotacji @Field pozwala na udostępnienie zadeklarowanej zmiennej metodom (funkcjom) skryptu. Taka zmienna staje się prywatnym polem (generowanej "pod spodem") klasy skryptu i dlatego jest dostępna w funkcjach (metodach) skryptu.

```
@Field def x
@Field String y
def z
int v
```

```
def func() {
  // x, y są widoczne
  // z, v - nie
}
```