

Programowanie obiektowe

Wykład 9

Marcin Młotkowski

21 kwietnia 2016

Plan wykładu

- 1 Typowanie w Ruby
- 2 Moduły i miksiny
 - Moduły
 - Miksiny
- 3 Bloki kodu
 - Deklarowanie bloków
 - Wykonywanie bloków
 - Bloki jako obiekty
 - Domknięcia
- 4 Projekt końcowy

Przypomnienie

Typowanie w w Ruby jest dynamiczne.

Duck typing (kacze typowanie)

"Jeśli chodzi jak kaczka i kwacze jak kaczka, to musi być kaczka"

Zastosowanie w programowaniu obiektowym

Jeśli obiekt ma odpowiednie metody, to jest taki jak trzeba.

Przykład w Javie

```
interface Kaczka
{
    String kwacz();
}
class Gęgawa implements Kaczka
{
    ...
}
```

Przykład w Ruby

```
class Cyraneczka
  def kwacz
    puts "kwa kwa"
  end
end
```

Przykład w Ruby

```
class Cyraneczka
  def kwacz
    puts "kwa kwa"
  end
end

def kwakanie(ptak)
  ptak.kwacz if ptak.respond_to? :kwacz
end
```


Przykład w Ruby

```
class Cyraneczka
  def kwacz
    puts "kwa kwa"
  end
end

def kwakanie(ptak)
  ptak.kwacz if ptak.respond_to? :kwacz
end

kwakanie(Cyraneczka.new)
kwakanie(5)
```

Plan wykładu

- 1 Typowanie w Ruby
- 2 **Moduły i miksiny**
 - Moduły
 - Miksiny
- 3 Bloki kodu
 - Deklarowanie bloków
 - Wykonywanie bloków
 - Bloki jako obiekty
 - Domknięcia
- 4 Projekt końcowy

Przykład definicji modułu

```
module Matematyka
  def Matematyka.dodawanie(x, y)
    x+y
  end
  Pi = 3.1415
end
```

Wykorzystanie modułu

```
require "modul"
```

```
puts Matematyka.dodaj(2, 2)
```

```
puts Matematyka::Pi
```

Import modułu

- `require plik` ładuje plik tylko raz, za pierwszym razem gdy sterowanie dochodzi do tej instrukcji;
- `load plik` ładuje pliki za każdym razem, gdy wykonanie programu dojdzie do tej instrukcji

Parę uwag o modułach

- Nazwa modułu musi być pisana wielką literą;
- w jednym pliku może być wiele modułów;
- moduły można zagnieżdżać;
- w module można umieszczać instrukcje, które są wykonywane podczas włączania modułu.

Domieszkowanie (mix-in) klas

Mechanizm włączania (wmiksowania) kodu modułu do klasy.

Przykład

Zadanie

Dla potrzeb logowania zdarzeń i debugingu jesteśmy zainteresowani, aby każdy obiekt umiał zwrócić migawkę swojego stanu, tj. wartości swoich pól.

Narzędzia

Refleksje (introspekcje): proces, podczas którego program może sam siebie obserwować i modyfikować.

Implementacja modułu

```
module Debugger
  def snapshot
    puts "Stan obiektu klasy #{self.class}"
    for iv in self.instance_variables
      puts "#{iv} = #{self.instance_variable_get(iv)}"
    end
  end
end
```

Wmiksowanie kodu

```
class DowolnaKlasa  
  include Debugger  
  ...  
end
```

Wmiksowanie kodu

```
class DowolnaKlasa  
  include Debugger  
  ...  
end
```

```
dk = DowolnaKlasa.new  
dk.snapshot
```

Inne zastosowania

Porównywanie obiektów

Moduł Comparable

- implementuje operatory porównania `<`, `<=`, `==`, `>=`, `>` i metodę `between?`
- wymaga implementacji operatora `<=>`

Przykład

```
class Wektor
  include Comparable
  def <=>(aWektor)
    ...
  end
end
```

Przykład

```
class Wektor
  include Comparable
  def <=>(aWektor)
    ...
  end
end
```

```
w1 = Wektor.new([3, -4, 5])
w2 = Wektor.new([-5, 12, -2])
w1 < w1
w1 >= w2
```

Plan wykładu

- 1 Typowanie w Ruby
- 2 Moduły i miksiny
 - Moduły
 - Miksiny
- 3 **Bloki kodu**
 - Deklarowanie bloków
 - Wykonywanie bloków
 - Bloki jako obiekty
 - Domknięcia
- 4 Projekt końcowy

Przypomnienie

Iteracja po kolekcjach:

```
[ 2, 3, 5, 7, 11].each { | val | print val, " " }
```

Co to jest blok

- blok to jest fragment kodu;
- blok może być obiektem (w końcu wszystko jest obiektem).

Definiowanie bloków — konwencje

Bloki bezparametrowe

Bloki jednowierszowe

```
{ puts "Hello"}
```

Bloki wielowierszowe

```
do
```

```
  instrukcja1
```

```
  instrukcja2
```

```
  instrukcja3
```

```
end
```

Bloki z parametrem

Bloki jednowierszowe

```
{ |x, y| puts "#{x} + #{y} daje #{x + y}" }
```

Bloki wielowierszowe

```
do |zm1, zm2|  
  instrukcja1  
  instrukcja2  
  instrukcja3  
end
```

Przetwarzanie kolekcji

```
[ "czerwony", "biały", "zielony"].each { | kolor | print kolor, " " }
```

Przetwarzanie kolekcji

```
[ "czerwony", "biały", "zielony"].each { | kolor | print kolor, " "}
```

```
5.times { puts "*"}
```

```
5.times { | i | print i, " "}
```

Przetwarzanie kolekcji

```
[ "czerwony", "biały", "zielony"].each { | kolor | print kolor, " " }
```

```
5.times { puts "*" }
```

```
5.times { | i | print i, " " }
```

```
('a'..'z').each { | znak | print znak }
```

Przetwarzanie kolekcji

```
[ "czerwony", "biały", "zielony"].each { | kolor | print kolor, " " }
```

```
5.times { puts "*" }
```

```
5.times { | i | print i, " " }
```

```
('a'..'z').each { | znak | print znak }
```

```
[ "H", "A", "L"].collect { | x | x.succ } → [ "I", "B", "M"]
```


Instrukcja **yield**

Deklarowanie funkcji

```
def run  
  yield  
end
```

Instrukcja **yield**

Deklarowanie funkcji

```
def run  
  yield  
end
```

Wywołanie funkcji

```
run { print "dwa dodać dwa jest ", 2 + 2 }
```

Deklarowanie funkcji

```
def run
  puts "Zaraz się zacznie\n"
  yield
  yield
  yield
  puts "Już się skończyło\n"
end
```

Deklarowanie funkcji

```
def run
  puts "Zaraz się zacznie\n"
  yield
  yield
  yield
  puts "Już się skończyło\n"
end
```

Wywołanie funkcji

```
run { print "dwa dodać dwa jest ", 2 + 2 }
```

Bloki z parametrem

```
def dodawanie  
  yield 2,2  
  yield 3,4  
end
```

Bloki z parametrem

```
def dodawanie  
  yield 2,2  
  yield 3,4  
end
```

```
dodawanie { | x, y | puts x + y }
```

Wiele argumentów

```
def foo(x, y, &blok)
  print x + y
  yield
end
```

Wiele argumentów

```
def foo(x, y, &blok)
  print x + y
  yield
end
```

```
foo(2, 3) { puts "A kuku" }
```


Inny przykład

```
def foo(x, y, &blok)
  yield x, y
end
```

Inny przykład

```
def foo(x, y, &blok)
  yield x, y
end
```

```
foo(2, 3) { | a, b | puts a + b }
```

Obiekty:

- można zapamiętać w zmiennej;
- wywoływać metody.

Tworzenie obiektów

Bloki mogą być obiektami klasy Proc. Metody tworzenia

- instrukcja `proc` ;
- instrukcja `lambda` ;
- `Proc.new blok`

Zalecane jest `lambda` .

Instrukcja `proc`

```
blok = proc { | x, y | puts x + y }  
blok.call(2, 3)
```

```
blok = proc do |x, y|  
  puts x + y  
end
```

```
blok.call(2, 3)
```

lambda

```
blok = lambda { puts 2+2 }
```

```
blok.call
```

```
blok = lambda do |x, y|  
  puts x + y  
end
```

```
blok.call(1,1)
```

Przykład

```
def powitanie(lang)
  lambda { |kto| lang + " " + kto }
end
```

Przykład

```
def powitanie(lang)
  lambda { |kto| lang + " " + kto }
end
```

```
ang = powitanie("Hello")
ang.call("Mr Bond") → "Hello Mr Bond"
```


Przykład

```
def powitanie(lang)
  lambda { |kto| lang + " " + kto }
end
```

```
ang = powitanie("Hello")
ang.call("Mr Bond") → "Hello Mr Bond"
```

```
pol = powitanie("Witaj")
pol.call("świecie") → "Witaj świecie"
```

Uwagi

Przypomnienie

```
def powitanie(lang)
  lambda { |kto| lang + " " + kto }
end
```

- blok został utworzony w kontekście ze zmienną lang
- kontekst "znika", a zmienna zostaje

Perwersja

```
def pudelko
  zawartosc = nil
  wez = lambda { zawartosc }
  wloz = lambda { |n| zawartosc = n }
  return wez, wloz
end
```

```
odczyt, zapis = pudelko
puts odczyt.call → nil
zapis.call(2)
puts odczyt.call → 2
```

Domknięcie

Domknięcie to funkcja wraz ze środowiskiem (tj. zmiennymi) w którym ta funkcja została utworzona.

Domknięcia jako obiekty

Obiekty klasy Proc mogą być przekazywane jak zwykłe argumenty.

Domknięcia jako obiekty

Obiekty klasy Proc mogą być przekazywane jak zwykłe argumenty.

Przykład

```
def bar(block, arg)
  puts block.call(arg)
end

bar(lambda { |n| n*n*n }, 10)
```

Proc.new *contra* lambda

```
def f1
  l = lambda { return "lambda"}
  l.call
  puts "Koniec f1"
end
```

```
def f2
  p = Proc.new { return "Proc"}
  p.call
  puts "Koniec f2"
end
```

f1

f2

Proc.new *contra* lambda

```
def f1
  l = lambda { return "lambda" }
  l.call
  puts "Koniec f1"
end
```

```
def f2
  p = Proc.new { return "Proc" }
  p.call
  puts "Koniec f2"
end
```

f1

f2

Wynik

Koniec f1

proc *contra* *lambda*

lambda wymaga dokładnie tylu argumentów ile zadeklarowano w bloku; *proc* (i *Proc.new*) ignoruje nadmiarowe argumenty, a brakującym nadaje wartość *nil*.

Jeszcze parę uwag

Kontrola przekazania bloku

```
def run
  if block_given?
    yield
  else
    puts "Brak bloku"
  end
end
```

Plan wykładu

- 1 Typowanie w Ruby
- 2 Moduły i miksiny
 - Moduły
 - Miksiny
- 3 Bloki kodu
 - Deklarowanie bloków
 - Wykonywanie bloków
 - Bloki jako obiekty
 - Domknięcia
- 4 Projekt końcowy

O czym ma być projekt

Co się chce ;)

W jakim języku programowania

Dowolny obiektowy, nie musi to być ani C[#], ani Java ani Ruby.

Wielkość projektu

Jak duży ma być program

Przynajmniej **siedem** zaimplementowanych klas.

Co się składa na projekt

Prezentacja na zajęciach

Predstawienie analizy obiektowej za pomocą diagramu klas.

Co się składa na projekt

Prezentacja na zajęciach

Predstawienie analizy obiektowej za pomocą diagramu klas.

Na koniec w sesji

- 1 Analiza obiektowa (pisemnie)
 - 1 spis klas, które implementuje program wraz z jednoakapitowym opisem roli klasy w systemie;
 - 2 diagram klas;
 - 3 użyte wzorce projektowe.
- 2 pliki źródłowe;
- 3 wersja skompilowana.