

EPI: Interfejs Graficzny 2011/2012

Wykład nr 3

Struktury sterujące języka

Aleksander Pohl

7 listopada 2011

Instrukcja warunkowa if

```
if (file_name =~ /\.rb$/) (then)
  lang = "ruby"
elsif (file_name =~ /\.pl$/) (then)
  lang = "perl"
else
  lang = "unknown"
end
```

if zwraca wartość ostatniego obliczonego wyrażenia

```
lang =
  if (file_name =~ /\.rb$/)
    "ruby"
  elsif (file_name =~ /\.pl$/)
    "perl"
  else
    "unknown"
  end
```

Instrukcja warunkowa if

then lub : wymagane w przypadku jednolinijkowców

```
lang =  
  if (file_name =~ /\.rb$/) then "ruby"  
  elsif (file_name =~ /\.pl$/) then "perl"  
  else "unknown"  
  end
```

```
lang =  
  if (file_name =~ /\.rb$/) : "ruby"  
  elsif (file_name =~ /\.pl$/) : "perl"  
  else "unknown"  
  end
```

Instrukcja warunkowa if,unless

```
unless (file_name =~ /\.rb$/)
  hint = "czemu nie Ruby?"
else
  hint = "dobry wybór!"
end
```

```
hint = (file_name =~ /\.rb$/ ? "dobry wybór!" : "czemu nie Ruby?")
```

if oraz unless jako modyfikatory:

```
puts "person = #{person}" if debug

puts person.name unless person.nil?
```

Instrukcja warunkowa – idiom Rubiego

Mamy zmienną „words”, którą chcemy wykorzystać jako tablicę słów. Chcemy dodać element, lecz nie wiemy, czy tablica została zainicjowana. Możemy to zrobić następująco:

```
if words.nil?  
  words = []  
end  
words << "new word"
```

Najlepiej jednak wykorzystać ten idiom Rubiego:

```
words ||= []  
words << "new word"
```

Algebra Boole'a

Jakie obiekty mają wartość true w Rubim? Wszystko co:

- ▶ nie jest wartością pustą (nil)
- ▶ nie jest fałszem (false)

0, pusty łańcuch, pusta tablica nie mają wartości „fałsz”

```
and &&  
or ||  
not !
```

```
if person && person.address  
  puts "#{person} #{person.address}"  
end
```

Deklaracje zmiennych

Badanie deklaracji zmiennych:

```
defined?(nie_ma_takiej_zmiennej)
```

```
#=> nil
```

```
nie_ma_takiej_zmiennej.nil?
```

```
#=> niezdefiniowana metoda lub zmienna lokalna
```

```
defined?(divider)
```

```
#=> nil
```

```
value = 10 / divider
```

```
#=> NameError: undefined local variable or method 'divider'
```

```
divider = nil
```

```
defined?(divider)
```

```
#=> "local-variable"
```

```
value = 10 / divider
```

```
#=> TypeError: nil can't be coerced into Fixnum
```

```
value = 10 / divider unless divider.nil?
```

Równość

- ▶ == „naturalna” równość
- ▶ eql? ten sam typ i identyczna wartość
- ▶ equal? to samo object_id
- ▶ < <= >= >
- ▶ <=> (-1,0,1)
- ▶ =~ dopasowanie wyrażeń regularnych
- ▶ === komparator w instrukcji case

(uwaga: symbole i liczby typu Fixnum posiadają zawsze tylko jedną instancję dla danej wartości)

Równość

```
0 == 0.0           #=> true
"ala" == "ala"     #=> true
[1,2,3] == [1,2,3] #=> true

1 == 1.0           #=> true
1.eql? 1.0         #=> false
"ala".eql? "ala"   #=> true

1.0.eql? 1.0       #=> true
1.0.equal? 1.0     #=> false
"ala".equal? "ala" #=> false
:ala.equal? :ala   #=> true

(1..2) === 1.5     #=> true
Fixnum === 1       #=> true
/^a/ === "ala"     #=> true
```

Równość – eql?

```
values = {}  
values[1] = "jeden"  
values[1.0] = "jeden"  
values  
#=> {1=>"jeden", 1.0=>"jeden"}
```

```
names = {}  
names["Ala"] = "Kowalska"  
names["Ala"] = "Smith"  
names  
#=> {"Ala"=>"Smith"}
```

Instrukcja selekcji

```
case month
when 1
  month_name = "styczeń"
when 2
  month_name = "luty"
when 3
  month_name = "marzec"
  #...
end
```

predykat `==` wykorzystywany jest do porównywania:

```
century =
  case year
  when 1901..2000
    "XX"
  when 2001..2100
    "XXI"
  else
    "ciemne wieki"
  end
```

Instrukcja selekcji

```
language =  
  case file_name  
  when /\.rb$/  
    "Ruby"  
  when /\.pl$/  
    "Perl"  
  when /\.java$/  
    "Java"  
  else  
    "Unknown"  
  end  
people =  
  case person  
  when String  
    [Person.new(:name => person)]  
  when Array  
    person  
  when Person  
    [person]  
  end
```

Bloki – problem

```
def even(tab)
  result = []
  for e in tab
    if e % 2 == 0 # tylko ta linia jest inna
      result << e
    end
  end
  result
end
def odd(tab)
  result = []
  for e in tab
    if e % 2 != 0 # tylko ta linia jest inna
      result << e
    end
  end
  result
end
```

Bloki – definicja

```
def three_times
  yield
  yield
  yield
end
```

```
three_times{puts "Hello!"}
# "Hello!"
# "Hello!"
# "Hello!"
```

```
three_times do
  puts "Hello!"
end
```

Bloki – przekazywanie wartości

```
def three_times
  yield 1
  yield 2
  yield 3
end
```

```
three_times{|i| puts "Hello! #{i}"}
# "Hello! 1"
# "Hello! 2"
# "Hello! 3"
```

```
def three_times
  puts yield(1)
  puts yield(2)
  puts yield(3)
end
```

```
three_times{|i| i * 5}
# 5
# 10
# 15
```

Bloki – rozwiązanie problemu

```
def select(tab)
  result = []
  for e in tab
    if yield(e)
      result << e
    end
  end
  result
end
```

```
select(tab){|e| e % 2 == 0}
```

```
select(tab){|e| e % 2 != 0}
```

select jest już zaimplementowany dla klasy Array

```
tab.select{|e| e % 2 == 0}
```

Bloki – block_given?

```
def open(file_name)
  file = File.open(file_name)
  if block_given?
    yield(file)
    file.close
  else
    file
  end
end

file = open("plik.txt")
# operacje na pliku
file.close

open("plik.txt") do |file|
  # operacje na pliku
end
```

Typy pętli

- ▶ loop
- ▶ while, until
- ▶ for
- ▶ times, upto, downto, step
- ▶ each

Pętla loop

```
loop do
  puts "pętla nieskończona :-)"
end
```

Kontrola pętli

- ▶ break – opuszcza aktualną pętlę
- ▶ redo – powtarza iterację bez sprawdzania warunku lub pobierania następnego elementu
- ▶ next – następna iteracja
- ▶ retry – powtarza całą pętlę (używać z rozważą)

```
loop do
  line = gets
  break if line.nil?
  puts line.upcase
end
```

Pętle while i until

`while` – wykonywana dopóki warunek jest prawdziwy

```
index = 1
while line = gets
  print "#{index}. #{line}"
  index += 1
end
```

`until` – wykonywana do czasu gdy warunek stanie się prawdziwy

```
value = 12345678
sum = 0
until value <= 0
  sum += (value / 10) % 10
  value /= 10
end
puts sum
```

Pętla for

Pętla for może być pętlą numeryczną:

```
for index in 10...30
  print "#{index}. "
end
```

Może również służyć do iterowania po strukturach sekwencyjnych:

```
for word in %w{Ala ma kota Mamrota}
  puts word
end
```

```
for number, letter in {1 => "A", 2 => "B", 3 => "C"}
  puts "#{number}. #{letter}"
end
```

Pętle „numeryczne”

`times` – określona ilość razy

```
3.times{ print "Ho! " } => "Ho! Ho! Ho! "
```

`upto` – od wartości minimalnej do maksymalnej

```
10.upto(20){|x| print x, " "} => "10 11 12 13 14 15 16 17 18 19 20"
```

`downto` – od wartości maksymalnej do minimalnej

```
20.downto(10){|x| print x, " "} => "20 19 18 17 16 15 14 13 12 11 10"
```

`step` – z określonym krokiem

```
0.step(100,10){|x| print x, " "} => "0 10 20 30 40 50 60 70 80 90 100 "
```

Pętla each

each – najczęściej wykorzystywana w roli pętli

```
(1..30).each do |index|  
  print "#{index}. "  
end
```

```
%w{Ala ma kota Mamrota}.each do |word|  
  puts word  
end
```

```
{1 => "A", 2 => "B", 3 => "C"}.each do |number,letter|  
  puts "#{number}. #{letter}"  
end
```

```
File.open("myfile.txt").each.with_index do |line,index|  
  puts "#{index + 1}. #{line}"  
end
```

Charakterystyka wyjątków

- ▶ używane są jako mechanizm obsługi błędów
- ▶ oferują alternatywną drogę przekazywania informacji do kodu nadrzędnego z wywoływanej funkcji
- ▶ pozwalają na obsługę błędów na dowolnym poziomie wywołania

Podjęcie tradycyjne

```
def open_file(file_name)
  # otwórz plik
  if File.exist?(file_name)
    # jeśli wszystko jest ok, przekaz deskryptor pliku
    File.open(file_name)
  else
    # jeśli nie, zwróć kod błędu
    -1
  end
end

def read_file(file_name, size)
  file = open_file(file_name)
  if file == -1
    puts "Nie można otworzyć pliku #{file_name}"
    return -1
  end
  result = file.read(size)
  file.close
  result
end
```

Podjęcie tradycyjne – cd.

```
result = read_file("plik1.txt",10)
if result == -1
  return
else
  puts result
end
result = read_file("plik2.txt",20)
if result == -1
  return
else
  puts result
end
result = read_file("plik3.txt",30)
if result == -1
  return
else
  puts result
end
```

Użycie wyjątków

```
def open_file(file_name)
  File.open(file_name)
  # może rzucić wyjątek, jeśli np. nie ma pliku
  # prawa do pliku są niewłaściwe, etc.
end

def read_file(file_name,size)
  file = open_file(file_name)
  # przekazuje na wyższy poziom wyjątek,
  # który może pojawić się w open_file
  result = file.read(size)
  # tutaj również może pojawić się wyjątek
  file.close
  # oraz tutaj
  result
end
```

Użycie wyjątków – cd.

```
begin
  puts read_file("plik1.txt",10)
  puts read_file("plik2.txt",20)
  puts read_file("plik3.txt",30)
rescue Exception => exception
  puts exception
  return
end
```

Obsługa wyjątków

```
begin
  # kod, który może rzucić wyjątek
rescue ExceptionType => exception
  # kod obsługi wyjątku
ensure
  # kod, który zostanie wykonany niezależnie
  # od wystąpienia, bądź niewystąpienia wyjątku
end

begin
  # kod, który może rzucić wyjątek
rescue ExceptionType1 => exception
  # kod obsługi wyjątku typu ExceptionType1
rescue ExceptionType2 => exception
  # kod obsługi wyjątku typu ExceptionType2
rescue
  # kod obsługi wyjątku StandardError
rescue Exception => exception
  # kod obsługujący wszystkie pozostałe wyjątki
end
```

Obsługa wyjątków – przykład

```
def file_operation(file_name)
  begin
    file = File.open(file_name)
    # operacje na pliku
  rescue Errno::ENOENT => exception
    puts "Plik #{file_name} nie istnieje"
  rescue Errno::EACCES => exception
    puts "Nie można odczytać pliku #{file_name}"
  rescue Exception => exception
    puts "Wystąpił nieoczekiwany problem " + exception
  end
end
```

Rzucanie wyjątków

```
raise "Invalid argument"  
# Błąd typu RuntimeError  
raise Exception.new("Invalid file name")  
# Błąd typu Exception  
raise  
# Błąd typu RuntimeError pozbawiony komunikatu  
  
begin  
  File.open(file_name)  
rescue Exception => exception  
  logger.error(exception)  
  # ponowne rzucenie przechwyconego wyjątku  
  raise  
end
```

Rzucanie wyjątków własnego typu

```
class MyException < Exception
end

raise MyException.new("Niepoprawna operacja")

begin
  # kod, który może rzucić wyjątek MyException
rescue MyException => exception
  # obsługa wyjątku MyException
end
```

Materiały

- ▶ Struktury sterujące w „Książce z kilofem”
ruby-doc.org/docs/ProgrammingRuby/html/tut_expressions.html
ruby-doc.org/docs/ProgrammingRuby/html/tut_exceptions.html
- ▶ Struktury sterujące w przewodniku po Rubim
apohllo.pl/dydaktyka/ruby/intro/struktury-jezyka

Pytania

PYTANIA?