

# EPI: Interfejs Graficzny 2009/2010

## Wbudowane typy danych

Aleksander Pohl

8 czerwca 2010

# Plan prezentacji

Łańcuchy

Liczby

Symbole i inne

W. regularne

Tablice i inne

Struktury języka

# Łańcuchy znaków

- ▶ sekwencje 8-bitowych bajtów (w wersji 1.9 pełne wsparcie dla Unicode oraz innych kodowań)
- ▶ ograniczane za pomocą ' lub "

- ▶ cytowanie apostrofów:

```
'test apostrophe: \''  
"test quote: \'"
```

- ▶ tylko łańcuchy ograniczone podwójnymi cudzysłowami mogą być 'interpolowane' za pomocą wyrażeń Rubiego:

```
name = 'john'  
"hello #{name}"  
'hello #{name}'
```

# Łańcuchy wielowierszowe

zaczynają się od znaków << i ciągu znaków, który ma symbolizować koniec łańcucha:

```
str = <<END_OF_STRING
    The body of the string
    is the input lines up to
    one ending with the same
    text that followed the '<<'  
END_OF_STRING
```

# Operacje na łańcuchach

„Mnożenie” i konkatelowanie:

```
"hello "+"world" #=> "hello world"  
"hello "*3 #=> 'hello hello hello "
```

```
a="hello"  
a << " world"  
a << "!!!" #=> "hello world!!!"
```

Badanie zawartości łańcucha:

```
"Test".empty? #=> false  
  
"hello".length #=> 5  
"hello".size #=> 5  
  
"hello".index('l') #=> 2  
"hello".rindex('l') #=> 3  
  
"hello"[0..3] #=> "hell"  
"hello"[-3..-1] #=> "llo"
```

# Operacje na łańcuchach

Manipulowanie łańcuchami:

```
"223".to_i #=> 223
```

```
"223.5".to_i #=> 223
```

```
"223.5".to_f #=> 223.5
```

```
"FF".to_i(16) #=> 255
```

```
"101010".to_i(2) #=> 42
```

```
"hello! how are you?".split  
#=> ["hello!", "how", "are", "you?"]
```

```
"hello! how are you".split(/[!\?] ?/)  
#=> ["hello", "how are you"]
```

```
"too many      spaces".squeeze(" ")  
#=> "too many spaces "
```

```
"Test".downcase #=> "test"
```

```
"hello \n".chomp #=> "hello "
```

```
"hello...".gsub(/\.\/, '!') #=> "hello!!!"
```

```
"hello...".sub(/\.\/, '!') #=> "hello!.."
```

# Operacje na łańcuchach

```
" trailing and leading spaces ".strip  
#=> "trailing and leading spaces"
```

konwencje nazewnicze: operacje modyfikujące i niemodyfikujące

- ▶ strip - zwraca nowy łańcuch
- ▶ strip! - modyfikuje oryginalny łańcuch

łączenie operacji:

```
"HELLO WITH SPACES ".squeeze.strip.downcase
```

## Zadania

- ▶ Napisz funkcję o nazwie `conversion`, która zamienia liczbę zapisaną w jednym systemie pozycyjną, na liczbę w innym systemie pozycyjnym. Funkcja powinna przyjmować trzy argumenty: pierwszy to łańcuch znaków reprezentujący liczbę podlegającą konwersji; drugi - to podstawa systemu, w którym zapisana jest przekazana liczba; trzeci - to podstawa systemu, w którym ma zostać zwrócony wynik.  
`conversion("100", 10, 16) => "64"`

## Zadania cd.

- ▶ Napisz funkcję `hex_adder`, która akceptuje dwa argumenty będące łańcuchami znaków reprezentującymi liczby szesnastkowe i jako wynik zwraca łańcuch znaków będący reprezentacją szesnastkową sumy tych liczb:  
`hex_adder("A", "B") => "15"`.
- ▶ Napisz funkcję `cleaner`, która w przekazanym jej łańcuchu znaków usuwa ostatni znak, jeśli jest znakiem nowej linii, usuwa spacje występujące na jego początku i końcu (po usunięciu znaku nowej linii) oraz zamienia litery na małe:  
`cleaner(" aAAAA a") => "aaaaa a"`.

# Plan prezentacji

Łańcuchy

Liczby

Symbole i inne

W. regularne

Tablice i inne

Struktury języka

# Liczby

- ▶ liczby całkowite – obiekty klasy Fixnum lub Bignum
- ▶ liczby zmiennopozycyjne – obiekty klasy Float

zamiana typu jest automatyczna:

```
a = 5           #=> 5
a.class        #=> Fixnum
a = a + 1.0    #=> 6.0
a.class        #=> Float
```

# Liczby

ale może prowadzić do niespodzianek:

```
a = 5
a / 2    #=> 2
a / 2.0  #=> 2.5
```

jawna konwersja typów:

```
7.5.to_i    #=> 7
8.to_f      #=> 8.0
8.5.to_s    #=> "8.5"
8.to_s(2)   #=> "1000"
15.to_s(16) #=> "f"
```

# Podstawowe operacje na liczbach

Przypisanie:

`a = 2`

`a,b = 3,4`

Operacje arytmetyczne:

<code>+</code>	dodawanie
<code>-</code>	odejmowanie
<code>*</code>	mnożenie
<code>/</code>	dzielenie
<code>%</code>	dzielenie modulo
<code>**</code>	potęgowanie

# Podstawowe operacje na liczbach

Obiektowa interpretacja operacji arytmetycznych:

```
2 + 3 * 7 #=> 2. + (3. * (7))
```

Operatory zachowują swoje naturalne priorytety (np. mnożenie wykonywane jest przed dodawaniem).

Operatory porównania: `<`, `<=`, `==`, `>=`, `>`

`<=>` komparator:

```
2 <=> 2 #=> 0
```

```
2 <=> 3 #=> -1
```

```
2 <=> 0 #=> 1
```

# Liczby jako obiekty

```
0.zero? #=> true
```

```
0.nonzero? #=> false
```

```
b = 5.5
```

```
b.round #=> 6
```

```
c = -7
```

```
c.abs #=> 7
```

```
10.between?(12,15) #=> false
```

```
2.5.integer? #=> false
```

```
7.succ #=> 8
```

# Zadania

- ▶ Napisać funkcję `sum` przyjmującą dwa argumenty, zwracającą jako rezultat sumę argumentów: `sum(1,2) => 3`.
- ▶ Napisz funkcję o nazwie `math1`, która dla przekazanego jej argumentu zwraca następnik z wartości bezwzględnej jej zaokrąglenia: `math1(-9.5) => 11`.
- ▶ Napisz funkcję o nazwie `divide`, która zwraca wynik dzielenia całkowitego jej pierwszego argumentu przez argument drugi. Jeśli do funkcji przekazane zostaną wartości ułamkowe, to powinna być brana największa wartość całkowita, nie większa od przekazanych wartości (tzw. podłoga): `divide(7.5,2.5) => 3`.

# Plan prezentacji

Łańcuchy

Liczby

**Symbole i inne**

W. regularne

Tablice i inne

Struktury języka

# Symbole

Symbol reprezentuje w Rubim pewną nazwę.  
Jest tworzony automatycznie poprzez użycie :dwukropka na początku nazwy:

```
:ruby
```

Tylko jeden obiekt klasy Symbol o danej nazwie jest tworzony w ciągu całego wykonania programu:

```
f1,f2 = :ruby,:ruby  
f1.object_id  
f2.object_id #ten sam obiekt
```

```
f1,f2 = "ruby","ruby"  
f1.object_id  
f2.object_id #dwa różne obiekty
```

# Zakresy

```
1..10 (zawiera 10)
1...10 (nie zawiera 10)
a = 1..10
a.min #=> 1
a.max #=> 10
a.include?(10) #=> true
```

jako sekwencje:

```
(1..10).to_a #=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

jako interwały:

```
(1..10) === 5 #=> true
(1..10) === 3.1415 #=> true
```

# Plan prezentacji

Łańcuchy

Liczby

Symbole i inne

W. regularne

Tablice i inne

Struktury języka

# Wyrażenia regularne

- ▶ obiekty klasy Regexp
- ▶ zapisywane z wykorzystaniem ukośników, np. `/hel.*`
- ▶ definiują wzorce dopasowywane do łańcuchów
- ▶ wszystkie znaki dopasowywane są do siebie samych, z wyjątkiem znaków specjalnych: `.`, `|`, `(`, `)`, `[`, `{`, `+`, `^`, `$`, `*`, `?`
- ▶ aby dopasować znak specjalny, należy poprzedzić go odwrotnym ukośnikiem

`/hel/` – dopasowuje „hel” w dowolnym miejscu łańcucha

```
/hel/ =~ "hello" #=> 0 - indeks początku dopasowania
```

```
/hel/ =~ "goodbye" #=> nil - brak dopasowania
```

```
/^hel/ #dopasuj tylko na początku
```

```
/hel$/ #dopasuj tylko na końcu
```

# Operacje na wyrażeniach regularnych

powtórzenia:

jeśli  $r$  jest aktualnym znakiem, to:

$r^*$  dopasowuje zero lub więcej wystąpień  $r$

$r^+$  dopasowuje jedno lub więcej wystąpień  $r$

$r^?$  dopasowuje zero lub jedno wystąpienie  $r$

$r\{m,n\}$  dopasowuje co najmniej „ $m$ ” i co najwyżej „ $n$ ” wystąpień  $r$

$r\{m,\}$  dopasowuje co najmniej „ $m$ ” wystąpień  $r$

alternatywa:

$|$  dopasowuje jedną z alternatyw

`/hello|goodbye/`

# Operacje na wyrażeniach regularnych

Klasy znaków:

[ ] – dopasowuje dokładnie jeden ze znaków wymienionych w nawiasie

Skróty klas znaków

<code>\d</code>	<code>[0-9]</code>	cyfry
<code>\D</code>	<code>[^0-9]</code>	nie-cyfry
<code>\s</code>	<code>[\s\t\r\n\f]</code>	białe spacje
<code>\S</code>	<code>[^\s\t\r\n\f]</code>	nie-białe spacje
<code>\w</code>	<code>[A-Za-z0-9_]</code>	znaki alfanumeryczne
<code>\W</code>	<code>[^A-Za-z0-9_]</code>	znaki nie-alfanumeryczne

# Operacje na wyrażeniach regularnych

Przykłady:

```
/hel.*/      #dopasowuje "hello"  
/hello!?!*/ #dopasowuje "hello" i "hello!"  
/hello!+*/  #dopasowuje "hello!!!!!!", ale nie "hello"  
/hello[!\.]/ #dopasowuje 'hello!' i 'hello.'
```

grupowanie:

() – wszystko wewnątrz nawiasów traktowane jest jako pojedyncze wyrażenie

```
/hello (John|James)!/
```

# Operacje na wyrażeniach regularnych

## Przechwytywanie podgrup:

```
"123Jan Kowalski456".sub(/([a-zA-Z]+) ([a-zA-Z]+)/, "\\2 \\1")  
#=> "123Kowalski Jan456"  
match_data = /([a-zA-Z]+) ([a-zA-Z]+)/.match("123Jan Kowalski456")  
#=> #<MatchData:0xb7aa6b8c>  
match_data[0] #=> "Jan Kowalski", inaczej $0  
match_data[1] #=> "Jan", inaczej $1  
match_data[2] #=> "Kowalski", inaczej $2  
match_data.pre_match #=> "123", inaczej $('  
match_data.post_match #=> "456", inaczej $'
```

## Zadania

- ▶ Napisać funkcję `words_swap`, która akceptuje jeden argument będący napisem. Funkcja powinna zamieniać miejscami pierwsze i trzecie słowo występujące w napisie. Słowa pisane są małymi literami alfabetu łacińskiego i oddzielone pojedynczymi spacjami.

```
words_swap("ala kot zenon") => "zenon kot ala"
```

- ▶ Napisz funkcję `proto_validator`, która akceptuje jeden argument będący łańcuchem znaków. Funkcja ta powinna zwracać wartość `true` (tzn. różną od `false` oraz `nil`) jeśli łańcuch znaków rozpoczyna się od poprawnego określenia protokołu w adresie URL. Poprawna nazwa protokołu składa się od 3 do 5 małych liter, po których następuje sekwencja `://`
- ```
proto_validator("http://ala.ma.kota.pl") && true  
=> true
```

## Zadania cd.

- ▶ Napisz funkcję `postal_code`, która akceptuje jeden argument będący napisem. Funkcja powinna zwracać wartość `true` (tzn. różną od `false` oraz `nil`), jeśli napis składa się z poprawnego kodu pocztowego, po którym następuje nazwa miasta pisana z wielkiej litery. W łańcuchu nie występują polskie znaki. Poprawny napis nie może zawierać spacji, ani innych białych znaków na początku, ani na końcu, a nazwa miasta musi składać się z co najmniej 2 liter.

```
postal_code("44-100 Gliwice") && true => true
```

# Plan prezentacji

Łańcuchy

Liczby

Symbole i inne

W. regularne

Tablice i inne

Struktury języka

# Tablice

```
arr = ["fred", 1, 3.14]
arr[0] #=> "fred"
arr[1] #=> 1
arr[-1] #=> 3.14
arr[-2] #=> 1
arr[0..1] #=> ["fred", 1]
arr[-2..-1] #=> [1, 3.14]
```

```
arr[0] = "Wilma"
arr #=> ["Wilma", 1, 3.14]
```

```
Array.new #=> []
Array.new(3) #=> [nil, nil, nil]
Array.new(3, "a") #=> ["a", "a", "a"]
```

zwięzły zapis tablic zawierających łańcuchy:

```
arr = %w(fred wilma barney betty the\ flintstones)
#=> ["fred", "wilma", "barney", "betty", "the flintstones"]
```

# Operacje na tablicach

```
a = [1,2,3]
```

```
b = [3,4]
```

```
a + b #=> [1,2,3,3,4]
```

```
a - b #=> [1,2]
```

```
a | b #=> [1,2,3,4]
```

```
a & b #=> [3]
```

```
a = [2,3]
```

```
a << 2 #=> [2,3,2]
```

```
a.push(5) #=> [2,3,2,5]
```

```
a.unshift(1) #=> [1,2,3,2,5]
```

```
a.index(2) #=> 1
```

```
a.rindex(2) #=> 3
```

# Operacje na tablicach

```
a = [1,2,3,4]
```

```
a.length #=> 4
```

```
a.empty? #=> false
```

```
a.reverse #=> [4,3,2,1]
```

```
a.first #=> 1
```

```
a #=> [1,2,3,4]
```

```
a.shift #=> 1
```

```
a #=> [2,3,4]
```

```
a.last #=> 4
```

```
a #=> [2,3,4]
```

```
a.pop #=> 4
```

```
a #=> [2,3]
```

# Operacje na tablicach

```
a = ["a", nil, "b", "b", nil, "c"]  
  
a.compact #=> ["a", "b", "b", "c"]  
a #=> ["a", nil, "b", "b", nil, "c"]  
a.compact! #=> ["a", "b", "b", "c"]  
a #=> ["a", "b", "b", "c"]  
a.uniq! #=> ["a", "b", "c"]  
  
a.join(", ") #=> "a, b, c"  
  
a.delete("a") #=> "a"  
a #=> ["b", "c"]  
a.delete_at(1) #=> "c"  
a #=> ["b"]  
  
a == ["b"] #=> true
```

# Tablice asocjacyjne

```
h = {"name" => "Fred", "surname" => "Flinstone"}  
h["name"]
```

```
h["name"] = "Wilma"  
h[:name] = "Wilma"
```

```
h1 = Hash.new  
h1[:foo]  
h2 = Hash.new(0)  
h2[:foo]
```

# Tablice asocjacyjne

```
h = {:foo => "bar", :bar => "baz"}

h.empty? #=> false

h.size #=> 2
h.length #=> 2

h.include?(:foo) #=> true
h.has_key?(:foo) #=> true #synonim include?
h.has_value?("bar") #=> true
h.index("bar") #=> :foo

h.keys #=> [:bar, :foo]
h.values #=> ["baz", "bar"]

h.delete(:foo)
h.clear
```

# Algebra Boole'a

Jakie obiekty mają wartość `true` w Rubim? Wszystko co:

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

0 nie ma wartości „fałsz”

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

Badanie deklaracji zmiennych:

```
defined?(dummy) #=> nil  
dummy.nil? #=> niezdefiniowana metoda lub zmienna lokalna
```

# Równość

- ▶ == równość
- ▶ === komparator w instrukcji case (switch)
- ▶ < <= >= >
- ▶ <=> (-1,0,1)
- ▶ =~ dopasowanie wyrażeń regularnych
- ▶ eql? ten sam typ i identyczna wartość
- ▶ equal? to samo object\_id (uwaga: symbole i liczby typu Fixnum posiadają zawsze tylko jedną instancję dla danej wartości)

```
m, n = 1, 1.0  
m == n #=> true  
m.eql? n #=> false
```

```
m, n = 1.0, 1.0  
m.eql? n #=> true  
m.equal? n #=> false
```

# Plan prezentacji

Łańcuchy

Liczby

Symbole i inne

W. regularne

Tablice i inne

Struktury języka

# Warunki

```
if (ext == "rb") (then)
  lang="ruby"
elsif (ext == "pl") (then)
  lang="perl"
else
  lang="unknown"
end
```

if zwraca wartość ostatniego obliczonego wyrażenia, więc

```
lang = if (ext == "rb")
  "ruby"
elsif (ext == "pl")
  "perl"
else
  "unknown"
end
```

# Warunki

then lub : wymagane w przypadku jednolinijkowców

```
lang = if (ext == "rb") then "ruby"  
elseif (ext == "pl") then "perl"  
else "unknown"  
end
```

```
lang = if (ext == "rb") : "ruby"  
elseif (ext == "pl") : "perl"  
else "unknown"  
end
```

# Warunki

```
unless (ext == "rb")
  hint = "go for ruby!"
else
  hint = "good choice!"
end
```

```
hint = ext=="rb" ? "good choice!" : "go for ruby"
```

if oraz unless jako modyfikatory:

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

```
return unless go_on
```

## Warunki – 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"
```

## Warunki – case

```
case ext
  when "rb" then lang="ruby"
  when "pl" then lang="perl"
  else lang="unknown"
end
```

operator `===` wykorzystywany jest do porównywania:

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

# Pętle

`while` – wykonywana dopóki warunek jest prawdziwy

```
while line = gets
  print line
end
```

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

```
count=100
until count < 10
  count-=1
end
```

# Pętle

while i until jako modyfikatory:

```
a=0  
puts a+=1 while a<10  
puts a-= until a<8
```

# Bloki

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

# Bloki

```
def three_times
  yield
  yield
  yield
end

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

three_times do
  puts "Hello!"
end
```

# Bloki

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

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

# Bloki

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

```
three_times{|i| puts "Hello! #{i}"; "#{i} pozdrowienia z bloku"}
# "Hello! 1"
# "1 pozdrowienia z bloku"
# "Hello! 2"
# "2 pozdrowienia z bloku"
# "Hello! 3"
# "3 pozdrowienia z bloku"
```

# Bloki

```
def fancy_format(str)
  if block_given?
    yield str
  else
    "~#{str}~"
  end
end

fancy_format("hello") #=> "~hello~"
fancy_format("hello"){|s| "####{s}###"} #=> "###hello###"

s=1
fancy_format("hello") do |s|
  "----#{s}----"
end
s => "hello"
```

# Iteratory

```
3.times{print "Ho! "} => "Ho! Ho! Ho! "  
  
0.upto(9){|x| print x," "} => "0 1 2 3 4 5 6 7 8 9 "  
  
0.step(12,3){|x| print x," "} => "0 3 6 9 12 "  
  
[1,2,3,4,5].each{|e| print e," "  
  
File.open("myfile.txt").each do |line|  
  #...  
end  
  
loop do  
  #...  
end
```

# Iteratory

```
for e in ['fee', 'fie', 'foe', 'fum']  
  print e  
end
```

```
['fee', 'fie', 'foe', 'fum'].each do |e|  
  print e  
end
```

```
for i in 0..9  
  print i+1  
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)

do `break`, `redo` i `next` można przekazać wartość:

```
result = while line = gets
  break(line) if line =~ /\d/
end
```

## Materiały źródłowe

- ▶ wbudowane typy Rubiego ruby-doc:
  - ▶ <http://www.ruby-doc.org/core/>
  - ▶ <http://www.ruby-doc.org/stdlib/>
- ▶ wbudowane typy w „książce z kilofem”:
  - ▶ <http://www.rubycentral.com/pickaxe/builtins.html>
  - ▶ [http://www.rubycentral.com/pickaxe/lib\\_standard.html](http://www.rubycentral.com/pickaxe/lib_standard.html)
- ▶ gotapi.com – nie tylko Ruby <http://www.gotapi.com>
- ▶ „Why’s poignant guid to Ruby” – nieco inne spojrzenie na język ;-)
- ▶ Typy standardowe, struktury języka  
<http://www.apohllo.pl/dydaktyka/ruby/intro>

## Materiały źródłowe

- ▶ Typy standardowe [http://www.ruby-doc.org/docs/ProgrammingRuby/html/tut\\_stdtypes.html](http://www.ruby-doc.org/docs/ProgrammingRuby/html/tut_stdtypes.html)
- ▶ Kontenery, bloki [http://www.ruby-doc.org/docs/ProgrammingRuby/html/tut\\_containers.html](http://www.ruby-doc.org/docs/ProgrammingRuby/html/tut_containers.html)
- ▶ Wyrażenia [http://www.ruby-doc.org/docs/ProgrammingRuby/html/tut\\_expressions.html](http://www.ruby-doc.org/docs/ProgrammingRuby/html/tut_expressions.html)
- ▶ Zmienne, wartości, obiekty  
<http://talklikeaduck.denhaven2.com/articles/2006/09/13/on-variables-values-and-objects>