

Wstęp do ruby dla programistów javy

czyli dlaczego java ssie

Błażej Święicki

Akademickie Stowarzyszenie Informatyczne


26 lutego 2011

Ruby vs Java

- Wieloparadygmatowy
- Imperatywny
- Typowanie
 - Silne
 - Dynamiczne
- Otwarte klasy
- Interpretowany
- Wszystko jest obiektem
- Ma przeładowanie operatorów


- Zorientowana obiektowo
- W większości imperatywna
- Typowanie
 - Silne
 - Statyczne
- Zamknięte klasy
- Kompilowana
- Prymitywne typy nie są obiektami

Hello world!



```
puts "Hello world!"
```

- 1 linia
- 19 znaków



```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, World");  
    }  
}
```

- 5 linii
- 106 znaków

Klasy i metody



```
# Komentarz
class NazwaKlasy # Musi być z dużej litery
  # Domyślnie publiczne
  def metoda a, b='domyślna_wartość'
  end

  def zwróc_siebie
    # Jeśli nie ma return x, to
    # metody zawsze zwracają to
    # co zwraca ostatnia linia.
    # return x działa jak w javie.
    self
  end

  def self.metoda_statyczna
  end
end
```



```
// Komentarz
class NazwaKlasy
{
  public Typ metoda(Typ a, String b) {
    // Nie da się zrobić domyślnej wartości
    if(b == null) {
      b = "domyślna wartość";
    }
  }

  public NazwaKlasy zwróc_siebie() {
    return this;
  }

  public static Typ metoda_statyczna() {
  }
}
```

Klasy i metody



```
# Komentarz
class NazwaKlasy # Musi być z dużej litery
  def lista_arg *args
    # args jest tablicą
    # ale metode wywołuje sie
    # lista_arg 1,2,3,4,5
    # zamiast lista_arg [1,2,3,4,5]
  end

  protected # Dalej metody chronione
  def metoda_chroniona1
    # Wywołanie metody, nawiasy
    # nie są potrzebne
    metoda_chroniona2
  end

  def metoda_chroniona2
  end
end
```



```
// Komentarz
class NazwaKlasy
{
  public Typ lista_argumentow(Typ... args) {
  }

  protected Typ metoda_chroniona1() {
    metoda_chroniona2();
  }

  protected Typ metoda_chroniona2() {
  }
}
```

Tworzenie zmiennych



```
tablica = [1,2,3,4,5]
tablica = (1..5).to_a # to samo

slovník = {'klucz' => 'wartość',
           'klucz2' => 'wartość2'}

a,b,c = 1,2,3

obj = Object.new
```



```
// Powinien być ArrayList
int[] tablica = {1,2,3,4,5};

import java.util.HashMap;
HashMap<String, String> slovník =
    new HashMap<String, String>;
slovník.put('klucz', 'wartość');
slovník.put('klucz2', 'wartość2');

int a = 1, b = 2, c = 3;

Object obj = new Object();
```

If, else itp.



```
a = 42
b = nil
if a == 5
  puts "A jest równe 5"
elsif a == 23
  puts "A jest równe 23"
elsif a # równoznaczne z a != nil && a != false
  puts "A jest prawdziwe"
else
  puts "A jest fałszywe"
end

puts "B jest prawdziwe" if b
puts "B jest fałszywe" unless b
```



```
int a = 42;
int b = null;
if(a == 5) {
  System.out.println("A jest równe 5");
} else if(a == 23) {
  System.out.println("A jest równe 23");
} else if(a != null) {
  System.out.println("A jest prawdziwe");
} else {
  System.out.println("A jest fałszywe");
}

if(b != null)System.out.println("B prawdziwe");
if(b == null)System.out.println("B fałszywe");
```

Pętle i operacje na zbiorach



```
5.times do |i|
  puts i
end

tablica = [1,2,3,4,5]
tablica.each do |element|
  puts element
end

while "cokolwiek prawdziwego"
  puts "Nieskończona pętla!"
end

tablica.map { |e| e*2 } # => [2,4,6,8,10]

# To samo, ale zmienia zawartość tablicy
# zamiast zwracać nową.
tablica.map! { |e| e*2 }
```



```
for(int i = 0; i < 5; i++) {
  System.out.println(i);
}

int[] tablica = {1,2,3,4,5};
for(int element: tablica) {
  System.out.println(element);
}

while(true) {
  System.out.println("Nieskończona pętla!");
}

int[] nowaTablica = new int[10];
for(int e: tablica) {
  nowaTablica[i] = e*2;
}

for(int i = 0; i < tablica.length; i++) {
  int e = tablica[i];
  tablica[i] = e*2;
} // A jak to zrobić na kolekcjach?
```

Pętle i operacje na zbiorach



```
tablica = [1,2,3,4,5]
tablica.reduce { |suma, e| suma + e } # => 15
tablica.reduce(:+) # to samo krócej, też => 15
```

```
tablica.select { |e| e % 2 == 0 } # => [2,4]
```

Metody z nazwami kończącymi się !
najczęściej zmieniają sam obiekt,
zamiast zwracać nowy.
tablica.select! { |e| e % 2 == 0 }



```
int[] tablica = {1,2,3,4,5};
int suma;
for(int e: tablica) {
    suma += e;
}
```

```
import java.util.ArrayList;
ArrayList<int> nowaTablica = new ArrayList<int>;
for(int e: tablica) {
    if(e % 2 == 0) {
        nowaTablica.add(e);
    }
}
```

```
// Bezpośrednio się nie da, trzeba kombinować
ArrayList<int> tmp = new ArrayList<int>;
for(int e: tablica) {
    if(e % 2 == 0) {
        tmp.add(e);
    }
}
tablica = tmp.toArray();
```

Wyjątki



```
begin
  raise Blad, "message"
rescue Blad => e
  puts "blad #{e}"
rescue
  raise
else
  # wykona się tylko jeśli
  # nie będzie żadnych wyjątków
ensure
  puts 'zawsze się wykonam'
end

rzuc_wyjatkciem rescue 5 # => 5
```



```
try {
  throw new Blad("message");
} catch(Blad e) {
  System.out.println("blad" + e.toString());
} catch(Exception e) {
  throw e;
}
# nie ma odpowiednika else

finally {
  System.out.println('zawsze sie wykonam');
}
```

Domknięcia, czyli o co chodziło z pętlami

Domknięcia to anonimowe funkcje przekazywane jako argument do metod. Nie mają odpowiedników w javie, za to w ruby są często używane.

Ogólna składnia

```
jakas_metoda { |arg1, arg2, ...| cialo_funkcji }  
# lub  
jakas_metoda do |arg1, arg2, ...|  
  cialo_funkcji  
end
```

Od środka funkcji

```
def jakas_metoda  
  yield # wywołuje domknięcie  
end  
# lub  
def jakas_metoda &block  
  # w końcu wszystko jest obiektem,  
  # więc domknięcie też  
  block.call # wywołuje domknięcie  
end
```

- Notacja: :nazwa
- Każdy symbol o danej nazwie jest tym samym obiektem, co pozwala na szybsze porównywanie.
- Każdy stworzony symbol zostaje w pamięci do końca wykonania programu.
- Dobrze się ich używa jako klucze hashy, jeśli są hardcoded.

Typy zmiennych, gettery i settery

Typy zmiennych

`nazwa` zmienna lokalna
`@nazwa` zmienna instancji
`@@nazwa` zmienna klasowa
`$nazwa` zmienna globalna

gettery i settery

```
class Klasa
  attr_reader :nazwa # getter dla @nazwa
  attr_writer :nazwa # setter dla @nazwa
  attr_accessor :nazwa # getter + setter
end
# Przykład:
a = Klasa.new
a.nazwa = 5
puts a.nazwa
```

Stringi i regexy

Stringi

```
a = 2
"String z interpolacją: #{a + 3}" # => String z interpolacją: 5
'Zwykły string: #{a + 3}' # => Zwykły string: #{a + 3}
%Q{'' #{a + 2}} # => '' 5
%q{''' #{a + 2}} # => ''' #{a + 2}
"%s %d" % ['printf!', a] # => printf! 2
"%<asdf>s %<asdf>s %<qwer>d" % {:asdf => 'printf', :qwer => a + 3}
# => printf printf 5
```

Regexy

```
/asdf/ # => Regexp
if "qwertyasdfgh" =~ /w.*(as.+)/ # => Fixnum lub nil
  puts "$& zawiera cały pasujący string"
  puts "$1 zawiera pierwszy subpattern"
  puts '$2..$9 zawierałyby kolejne, gdyby były'
end
```

Suma liczb pierwszych ≤ 1000

Ruby – 79 znaków

```
(2..1000).reject { |n| (2..n-1).find { |x| n % x == 0 } }.reduce(:+) # => 76127
```

Java – 173 znaki

```
int suma = 0;
for(int i = 2; i <= 1000; i++) {
    bool found = false;
    for(int j = 2; j < i; j++) {
        if(i % j == 0) {
            found = true;
            break;
        }
    }
    if(!found) {
        suma += i;
    }
}
```

Sortowanie liczb zespolonych wg modułu

Ruby – 80 znaków

```
lista = [ 1 + 3.i, 5 + 1.i, 2 + 2.i ]  
puts lista.sort_by { |c| c.abs }.join ", "
```

Java – 1039 znaków

```
import java.util.Collections; import java.util.Comparator;  
import java.util.List; import java.util.ArrayList;  
class Complex  
{  
    private Double re;  
    private Double im;  
  
    public Double getRe() { return re; }  
    public void setRe(Double re) { this.re = re; }  
    public Double getIm() { return im; }  
    public void setIm(Double im) { this.im = im; }  
    public Double abs2() { return re * re + im * im; }  
    public Complex(Double re, Double im) { this.re = re; this.im = im; }  
}  
// ...
```

Sortowanie liczb zespolonych wg modułu, ciąg dalszy

Java – 1039 znaków

```
// ...
class ComplexModuleComparator implements Comparator
{
    public int compare(Object a, Object b) {
        return ((Complex)a).abs2().compareTo(((Complex)b).abs2());
    }
    public boolean equals(Object o) { return this == o; }
}
class ComplexSort
{
    public static void main(String args[]) {
        List<Complex> lista = new ArrayList<Complex>();
        lista.add(new Complex(1.0,3.0));
        lista.add(new Complex(5.0,1.0));
        lista.add(new Complex(2.0,2.0));
        Collections.sort(lista, new ComplexModuleComparator());
        for(Complex c: lista) {
            System.out.println(c.getRe().toString() + " + "
                + c.getIm().toString() + "i");
        }
    }
}
```

Zdefiniujmy klasę Complex w ruby

Ruby

```
class Complex
  attr_accessor :re, :im # definiuje gettery i settery
  def initialize re, im
    @re, @im = re, im
  end

  def abs
    Math.sqrt(re**2 + im**2)
  end

  def to_s #rzutowanie na string
    "#{@re} + #{@im}i"
  end
end

class Fixnum
  def i
    Complex(0, self)
  end
end
```

Quicksort, czyli wygoda dynamicznego typowania

Ruby – 218 znaków

```
class Array
  def qsort
    return self if length < 2
    pivot = self[length/2]
    select { |i| i < pivot }.qsort + select { |i| i == pivot } +
    select { |i| i > pivot }.qsort
  end
end
```

Java – 525 znaków

```
public static <E extends Comparable<? super E>> List<E> quickSort(List<E> list)
{
    if (list.size() <= 1)
        return list;

    E pivot = list.get(0);

    List<E> smaller = new LinkedList<E>();
    List<E> bigger  = new LinkedList<E>();
    List<E> same    = new LinkedList<E>();

    //...
```

Quicksort, czyli wygoda dynamicznego typowania

Java – 525 znaków

```
// ...
for (E i: list) {
    if (i.compareTo(pivot) < 0) {
        smaller.add(i);
    } else if (i.compareTo(pivot) > 0) {
        bigger.add(i);
    } else {
        same.add(i);
    }
}

smaller = quickSort(smaller);
bigger = quickSort(bigger);

smaller.addAll(same);
smaller.addAll(bigger);
return smaller;
}
```

`irb` interaktywny interpreter (czyli Read-Eval-Print Loop)

`ri` dokumentacja (Przykład: `ri Array`)

`gem` menedżer paczek

`rake` ruby make

Pytania?