

Metaprogramowanie w Ruby

Błażej Święicki

24 marca 2011

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

Moduły

Moduły pozwalają na łatwe włączanie ich kodu w klasy. Moduł definiuje się dokładnie tak jak klasę, tylko słowem kluczowym `module` zamiast `class`.

Mixiny

Mixin to po prostu moduł włączony w jakąś klasę, np. tak:

```
class Klasa
  include Enumerable
end
```

Hierarchia klas

```
obiekt.class  
obiekt.singleton_class
```

```
Klasa.ancestors  
# Czy jest instancją InnejKlasy?  
obiekt.instance_of? InnaKlasa  
# ... lub jej podklasy?  
obiekt.is_a? InnaKlasa
```

```
Klasa.class_eval &blok  
obiekt.instance_eval &blok
```

Jak się dostać do środka klasy – zmienne i stałe

```
obiekt.instance_variable_defined? :@nazwa
obiekt.instance_variable_get :@nazwa
obiekt.instance_variable_set :@nazwa, 'wartość'
obiekt.instance_variables
```

```
Klasa.class_variable_defined? :@@nazwa
Klasa.class_variable_get :@@nazwa
Klasa.class_variable_set :@@nazwa, 'wartość'
Klasa.class_variables
```

```
Klasa.const_defined? :Nazwa
Klasa.const_get :Nazwa
Klasa.const_set :Nazwa, 'wartość'
Klasa.remove_const :Nazwa
Klasa.constants
```

Jak się dostać do środka klasy – metody

```
# Zwraca metodę o nazwie :metoda
Klasa.instance_method :metoda
Klasa.instance_methods

Klasa.alias_method :nowa_metoda, :stara_metoda
# Ustawia widoczność metody na public, protected
# lub private
Klasa.public :metoda
Klasa.protected :metoda
Klasa.private :metoda
Klasa.method_defined? :metoda
```

Jak się dostać do środka klasy – metody

```
# Zwraca metodę o danej nazwie, przypisaną do obiektu
obiekt.method :metoda
# Wywołanie metody o nazwie :metoda
obiekt.send :metoda, *argumenty, &blok
# Czy można wywołać metodę?
obiekt.respond_to? :metoda
```

Imperatywność

Ruby jest w pełni imperatywny, co oznacza że nie ma w kodzie miejsc, w których nie można wykonać kodu.

Imperatywność

Ruby jest w pełni imperatywny, co oznacza że nie ma w kodzie miejsc, w których nie można wykonać kodu.

```
class Klasa
  2 + 3
end # => 5
```

Imperatywność

Ruby jest w pełni imperatywny, co oznacza że nie ma w kodzie miejsc, w których nie można wykonać kodu.

```
class Klasa
  2 + 3
end # => 5
```

Dlaczego w klasie?

Imperatywność

Ruby jest w pełni imperatywny, co oznacza że nie ma w kodzie miejsc, w których nie można wykonać kodu.

```
class Klasa
  2 + 3
end # => 5
```

Dlaczego w klasie?

Kod w środku klasy jest wykonywany w kontekście klasy.

```
class Klasa
  def self.return5
    2 + 3
  end
  return5
end # => 5
Klasa.return5 # => 5
```

Klasy, nawet te wbudowane, można dynamicznie modyfikować.

Klasy, nawet te wbudowane, można dynamicznie modyfikować.

```
class Fixnum
  alias_method :plus, :+
  def + b
    1.plus self.plus(b)
  end
end
2 + 2 # => 5
```

Wszystko jest obiektem

```
Object.is_a? Class # => true
Class.is_a? Object # => true
Object.class      # => Class
Class.class       # => Class
Class.ancestors   # => [Class, Module, Object,
                        Kernel, BasicObject]
```

Wszystko jest obiektem

```
Object.is_a? Class # => true
Class.is_a? Object # => true
Object.class      # => Class
Class.class       # => Class
Class.ancestors   # => [Class, Module, Object,
                        Kernel, BasicObject]
```

```
nil.class # => NilClass
nil.class.ancestors # => [NilClass, Object,
                        Kernel, BasicObject]
```

Uwaga

W ruby, "singleton" to nie jest nazwa na design pattern.

Uwaga

W ruby, "singleton" to nie jest nazwa na design pattern.

```
class << obiekt
  # Środek singletonu
end
singleton = class << obiekt; self; end
```

Uwaga

W ruby, "singleton" to nie jest nazwa na design pattern.

```
class << obiekt
  # Środek singletonu
end
singleton = class << obiekt; self; end
```

```
class Klasa
  class << self
    def metoda
# metoda statyczna Klasy
    end
  end
end
```

```
class Klasa
  def self.metoda
# metoda statyczna Klasy
  end
end
```

Dlaczego "singleton"?

```
a = Object.new
class << a
  def test
    5
  end
end
a.test # => 5
Object.new.test # NoMethodError
```

Dynamiczne definiowanie metod

```
class Class
  def attr_reader name
    define_method name do
      instance_variable_get "@#{name}"
    end
  end

  def attr_writer name
    define_method "#{name}=" do |value|
      instance_variable_set "@#{name}", value
    end
  end
end
```

Dynamiczne definiowanie metod – w singletonach

```
class Class
  def class_attr_reader name
    singleton_class.send(:define_method, name) do
      instance_variable_get "@#{name}"
    end
  end

  def class_attr_writer name
    singleton_class.send(:define_method, "#{name}=") do |value|
      instance_variable_set "@#{name}", value
    end
  end
end
```

Przechwytywanie informacji o tym, co się zmienia

Klasa.method_added name

Klasa.inherited by

Modul.included by

```
class Klasa
  def self.method_added name
    puts "Dodano metodę #{name}."
  end

  def self.inherited by
    puts "Nowa podklasa: #{by}."
  end
end
```

Wywołania nieistniejących metod

Gdy na obiekcie wywołana jest metoda, która nie jest zdefiniowana, ruby wywołuje:

```
Obiekt.method_missing nazwa, *argumenty, &blok Jej  
domyślna implementacja rzuca wyjątkiem NoMethodError.
```

Wywołania nieistniejących metod

Gdy na obiekcie wywołana jest metoda, która nie jest zdefiniowana, ruby wywołuje:

`Obiekt.method_missing nazwa, *argumenty, &blok` Jej domyślna implementacja rzuca wyjątkiem `NoMethodError`.

Przykład implementacji

```
class Klasa
  def method_missing name, *args
    super unless name.to_s.end_with? '!'
    puts "Wywołano metodę #{name} z " +
         "#{args.length} argumentami."
  end

  def respond_to? name; name.to_s.end_with? '!'; end
end
```

Przykład – Domain Specific Language

Przykład klasy, która pozwala łatwo tworzyć pliki xml.

```
class XML < BasicObject
  def method_missing sym, attr = {}, &block
    @output << "<#{sym}"

    attr.each do |key, value|
      @output << " #{key}='#{value}'"
    end

    if block
      @output << ">" << block.call.to_s << "</#{sym}>"
    else
      @output << " />"
    end
    nil
  end
end
```

Przykład – Domain Specific Language

```
class XML
  def initialize file = $stdout
    @output = file
  end

  def respond_to?
    true
  end

  def self.render file = $stdout, &block
    self.new(file).instance_eval &block
  end
end
```

Domain Specific Language – użycie

```
XML.render do
  html :xmlns => "http://www.w3.org/1999/xhtml" do
    head do
      title { "Hello World" }
    end
    body do
      h1 { "Hello World" }
      ul :type => "disc" do
        10.times do |i|
          li { i }
        end
      end
      nil
    end

    div :style => "font-family:serif" do
      "To jest przykładowy dokument XML"
    end
  end
end
end
end
```

Pytania