

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

FRAGMENTY KSIĄŻEK ONLINE

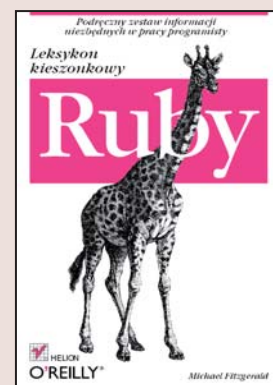
Ruby. Leksykon kieszonkowy

Autor: Michael Fitzgerald

ISBN: 978-83-246-1384-7

Tytuł oryginału: [Ruby Pocket Reference](#)

Format: B6, stron: 192



Podręczny zestaw informacji niezbędnych w pracy programisty

Ruby to obiektowy język programowania, opracowany w 1995 roku w Japonii. Dzięki swojej prostej składni, zwartej konstrukcji i sporym możliwościom błyskawicznie zyskał ogromne grono zwolenników. Pojawienie się mechanizmu Ruby on Rails, niesamowicie usprawniającego tworzenie aplikacji i witryn internetowych zgodnych z nurtem Web 2.0, dodatkowo zwiększyło popularność języka Ruby. W oparciu o ten język powstało wiele znanych serwisów WWW, odwiedzanych każdego dnia przez tysiące gości.

„Ruby. Leksykon kieszonkowy” to zestawienie niezbędnych informacji o tym języku, przydatne podczas codziennej pracy programisty. Znajdziesz w nim informacje o słowach kluczowych, operatorach, zmiennych i stałych. Przeczytasz także o formatowaniu tekstu, wyrażeniach regularnych, operacjach na plikach i programowaniu obiektowym. Dowiesz się ponadto, jak korzystać z interaktywnego Ruby i RDoc.

- Uruchamianie interpretera Ruby
- Słowa kluczowe
- Zmienne
- Instrukcje warunkowe
- Programowanie obiektowe
- Moduły
- Operacje na plikach
- Obsługa wyjątków
- Metody klas Array, Hash, Object, Kernel i String
- Wyszukiwanie i usuwanie błędów

**Usprawnij i przyspiesz swoją pracę,
korzystając z leksykonów kieszonkowych**



Spis treści

Uruchomienie Ruby	8
Uruchomienie interpretera Ruby	9
Wykorzystywanie #! w Uniksie oraz Linuksie	11
Skojarzenie rozszerzenia pliku w systemie Windows	11
Zarezerwowane słowa kluczowe	13
Operatory	16
Komentarze	17
Liczby	17
Zmienne	19
Zmienne lokalne	19
Zmienne instancji	19
Zmienne klasy	20
Zmienne globalne	20
Stałe	20
Równoległe przypisanie zmiennych	21
Symbole	21
Wbudowane zmienne	22
Pseudozmienne	25

Stałe globalne	26
Przedziały	27
Metody	27
Nawiasy	28
Zwracanie wartości	28
Konwencje nazewnictwa metod	29
Argumenty domyślne	30
Zmienna liczba argumentów	30
Aliasy metod	31
Bloki	31
Procedury	34
Instrukcje warunkowe	35
Instrukcja if	35
Instrukcja unless	38
Instrukcja while	38
Instrukcja until	40
Instrukcja case	41
Pętla for	42
Operator trójargumentowy	43
Wykonywanie kodu przed programem bądź po programie	43
Klasy	44
Zmienne instancji	45
Aksesory	47
Zmienne klasy	49
Metody klasy	49
Singletony	50
Dziedziczenie	51
Publiczna, prywatna czy chroniona	52
Moduły oraz mixiny	54

Pliki	56
Tworzenie nowego pliku	57
Otwieranie istniejącego pliku	58
ARGV oraz ARGF	58
Zmiana nazwy pliku oraz jego usunięcie	59
Badanie plików	59
Tryby oraz własność plików	60
Klasa IO	62
Obsługa wyjątków	64
Klauzule rescue oraz ensure	65
Metoda raise	65
Metody catch oraz throw	66
Klasa Object	66
Metody instancji klasy Object	67
Moduł Kernel	73
Klasa String	85
Podstawianie wyrażeń	85
Łańcuchy znaków z ogranicznikami	86
Dokumenty miejscowe	86
Znaki ucieczki	87
Kodowanie znaków	88
Wyrażenia regularne	89
Metody klasy String	95
Klasa Array	110
Tworzenie tablic	111
Metody klasy Array	113
Metody instancji klasy Array	114

Klasa Hash	125
Tworzenie tablic asocjacyjnych	125
Metody klasy Hash	127
Metody instancji Hash	127
Dyrektywy służące do formatowania czasu	132
Ruby interaktywny (irb)	133
Debugger języka Ruby	137
Dokumentacja Ruby	140
Opcje RDoc	142
RubyGems	147
Rake	152
Istniejące zasoby dla języka Ruby	155
Słowniczek	156
Skorowidz	177

Przedziały

Ruby obsługuje przedziały dzięki wykorzystaniu operatorów `..` (przedział domknięty) oraz `...` (lewostronnie domknięty, prawostronnie otwarty). Na przykład przedział `1..12` zawiera liczby 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12 (z 12 włącznie). W przedziale `1...12` wartość końcowa 12 zostaje jednak wykluczona, czyli w praktyce znajdują się w nim liczby 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11.

Metoda `===` sprawdza, czy wartość mieści się w przedziale:

```
(1..25) === 14 # => true, w przedziale
(1..25) === 26 # => false, poza przedziałem
(1...25) === 25 # => false, poza przedziałem (wykorzystano operator ...)
```

Można wykorzystać przedział na przykład do utworzenia tablicy cyfr:

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

Przedział można również utworzyć w następujący sposób:

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

Metody

Metody umożliwiają grupowanie instrukcji oraz wyrażeń programistycznych w jednym miejscu, by dało się je wykorzystywać w sposób wygodny i w miarę potrzeby — również powtarzalny. Większość operatorów z języka Ruby jest także metodami. Poniżej znajduje się prosta definicja metody o nazwie `hello` utworzonej z wykorzystaniem słów kluczowych `def` oraz `end`:

```
def hello
  puts "Witaj świecie!"
end

hello # => Witaj świecie!
```

Definicję metody można usunąć za pomocą undef:

```
undef hello # usuwa definicję metody o nazwie hello

hello # teraz należy spróbować wywołać tę metodę
NameError: undefined local variable or method 'hello' for
↳main:Object
  from (irb):11
  from :0
```

Metody mogą przyjmować argumenty. Zaprezentowana niżej metoda repeat przyjmuje dwa argumenty, word oraz times:

```
def repeat( word, times )
  puts word * times
end

repeat("Witaj! ", 3) #=> Witaj! Witaj! Witaj!
repeat "Do widzenia! ", 4 #=> Do widzenia! Do widzenia! Do widzenia!
↳Do widzenia!
```

Nawiasy

W większości definicji metod oraz wywołań w języku Ruby nawiasy są opcjonalne. Jeśli pominie się nawiasy przy wywoływaniu metody przyjmującej argumenty, w zależności od typów argumentów można otrzymać ostrzeżenia.

Zwracanie wartości

Metody zwracają wartości. W innych językach programowania wartości zwraca się w sposób jawny za pomocą instrukcji return. W języku Ruby wartość ostatniego obliczonego wyrażenia zwracana jest bez względu na fakt użycia jawnej instrukcji return. Jest to cecha charakterystyczna Ruby. Można również zdefiniować zwracaną wartość za pomocą słowa kluczowego return:

```
def hello
  return "Witaj świecie!"
end
```

Konwencje nazewnictwa metod

Ruby posiada konwencje dotyczące ostatniego znaku w nazwie metody — są one często spotykane w tym języku, jednak nie są wymuszane. Jeśli nazwa metody kończy się znakiem zapytania (?), jak w `eql?`, oznacza to, że metoda zwraca wartość typu Boolean — `true` bądź `false`. Na przykład:

```
x = 1.0
y = 1.0
x.eql? y #=> true
```

Jeśli nazwa metody kończy się wykrzyknikiem (!), jak `delete!`, oznacza to, że metoda jest destruktywna, czyli *wprowadza zmiany na miejscu do samego obiektu*, a nie jego kopii — zmienia więc oryginalny obiekt. Różnicę widać na przykładzie metod `delete` oraz `delete!` obiektu `String`:

```
der_mensch = "Matz!" #=> "Matz!"
der_mensch.delete( "!" ) #=> "Matz"
puts der_mensch #=> Matz!
der_mensch.delete!( "!" ) #=> "Matz"
puts der_mensch #=> Matz
```

Jeśli nazwa metody kończy się znakiem równości (=), jak w `family_name=`, oznacza to, że metoda jest typu *setter*, czyli wykonuje przypisanie bądź ustawia zmienną, taką jak zmienna instancji w klasie:

```
class Name
  def family_name=( family )
    @family_name = family
  end
  def given_name=( given )
    @given_name = given
  end
end

n = Name.new
n.family_name= "Matsumoto" #=> "Matsumoto"
n.given_name= "Yukihiro" #=> "Yukihiro"
p n #=> <Name:0x1d441c @family_name="Matsumoto", given_name="Yukihiro">
```


Argumenty domyślne

Zaprezentowana wcześniej metoda `repeat` zawierała dwa argumenty. Argumentom tym można nadać wartości domyślne poprzez użycie znaku równości, po którym następuje wartość. Kiedy wywoła się metodę bez argumentów, automatycznie wykorzystane zostaną wartości domyślne. Należy zdefiniować metodę `repeat` ponownie, dodając do niej wartości domyślne — `Witaj!` dla `word` oraz `3` dla `times`. Należy wywołać tę metodę najpierw bez argumentów, a następnie z nimi:

```
def repeat( word="Witaj!", times=3 )
  puts word * times
end

repeat #=> Witaj! Witaj! Witaj!

repeat( "Do widzenia! ", 5 ) #=> Do widzenia! Do widzenia! Do widzenia!
↳Do widzenia! Do widzenia!
```

Zmienna liczba argumentów

Ponieważ Ruby pozwala na przekazywanie do metody zmiennej liczby argumentów dzięki poprzedzeniu argumentu znakiem `*`, autor programu ma w tym zakresie dużą elastyczność:

```
def num_args( *args )
  length = args.size
  label = length == 1 ? " argument" : " argumentów"
  num = length.to_s + label + " ( " + args.inspect + " )"
  num
end

puts num_args #=> 0 argumentów ( [])

puts num_args(1) #=> 1 argument ( [1] )

puts num_args( 100, "witaj", 2.5, "trzy", 99009 ) #=> 5 argumentów
↳( [100, "witaj", 2.5, "trzy", 99009] )
```

Można również wykorzystać argumenty ustalone w połączeniu z argumentami zmieniającymi się:

```
def two_plus( one, two, *args )
  length = args.size
  label = length == 1? " argument zmieniający się" : "
argumentów zmieniających się"
  num = length.to_s + label + " ( " + args.inspect + " )"
  num
end

puts two_plus( 1, 2 ) #=> 0 argumentów zmieniających się( [1] )
puts two_plus( 1000, 3.5, 14.3 ) #=> 1 argument zmieniający się( [14.3] )
puts two_plus( 100, 2.5, "trzy", 70, 14.3, "witaj", 99009 )
↳#=> 5 argumentów zmieniających się (["trzy", 70, 14.3, "witaj", 99009])
```

Alias metody

Język Ruby posiada słowo kluczowe *alias*, które służy do tworzenia aliasów metod. *Alias* oznacza, że można utworzyć kopię metody z nową nazwą, choć obie metody odnosić się będą do tego samego obiektu. Poniższy przykład ilustruje sposób tworzenia aliasu dla metody `greet`:

```
def greet
  puts "Witaj, kochanie!"
end

alias baby greet # utworzenie aliasu baby dla metody greet

greet # wywołanie metody
Witaj, kochanie!

baby # wywołanie aliasu
Witaj, kochanie!
```

Bloki

Blok (ang. *block*) w Ruby jest czymś więcej niż tylko blokiem kodu bądź grupą instrukcji. Jest on zawsze wywoływany w połączeniu z metodą, co zostanie zaraz zaprezentowane. Tak naprawdę bloki

są domknięciami (ang. *closure*), czasami określanymi mianem *funkcji bezimiennych* (ang. *nameless function*). Są jak metoda znajdująca się wewnątrz innej metody, która współdzieli zmienne lub odnosi się do zmiennych z metody zewnętrznej. W języku Ruby domknięcie lub blok umieszczone są w nawiasach klamrowych ({}) lub też pomiędzy do oraz end, a ich działanie uzależnione jest od powiązanej z nimi metody (na przykład each).

Poniżej znajduje się przykład wywołania bloku metody each obiektu Array:

```
pacific = [ "Waszyngton", "Oregon", "Kalifornia" ]

pacific.each do |element|
  puts element
end
```

Nazwa znajdująca się pomiędzy znakami | (czyli |element|) może być dowolna. Blok wykorzystuje ją jako zmienną lokalną przechowującą każdy z elementów tablicy, a później używa jej, aby zrobić coś z tym elementem. Można zastąpić do oraz end nawiasami klamrowymi, jak się to często robi. Same nawiasy klamrowe mają tak naprawdę wyższy priorytet od konstrukcji z do oraz end:

```
pacific.each ( |e| puts e )
```

Jeśli użyje się nazwy zmiennej, która już istnieje w zakresie nadrzędnym, blok przypisuje zmiennej każdą kolejną wartość, co może nie być zgodne z zamierzeniami autora. Nie generuje zmiennej lokalnej dla bloku o tej nazwie, jak można by tego oczekiwać. W ten sposób otrzyma się następujące zachowanie:

```
j = 7
(1..4).to_a.each { |j| } #j jest teraz równe 4
```

Instrukcja yield

Instrukcja yield wykonuje blok powiązany z metodą. Metoda gimme zawiera na przykład jedynie instrukcję yield:

```
def gimme
  yield
end
```

Żeby dowiedzieć się, co robi `yield`, należy wywołać `gimme` i zobaczyć, co się stanie:

```
gimme
LocalJumpError: no block given
  from (irb):11:in 'gimme'
  from (irb):13
  from :0
```

Otrzymuje się błąd, ponieważ zadanie instrukcji `yield` polega na wykonaniu bloku kodu powiązanego z metodą. Tego właśnie brakowało w wywołaniu metody `gimme`. Można uniknąć tego błędu dzięki skorzystaniu z metody `block_given?` (z Kernel). Należy ponownie zdefiniować `gimme` z instrukcją `if`:

```
def gimme
  if block_given?
    yield
  else
    puts "Nie zawieram bloku!"
  end
end
```

Teraz można ponownie wypróbować metodę `gimme` z blokiem oraz bez niego:

```
gimme { print "Powiedz wszystkim 'cześć'." } #=> Powiedz
↳wszystkim 'cześć'.

gimme #=> Nie zawieram bloku!
```

Teraz należy ponownie zdefiniować metodę `gimme`, by zawierała ona dwie instrukcje `yield`, a następnie wywołać ją z blokiem:

```
def gimme
  if block_given?
    yield
    yield
  else
    puts "Nie zawieram bloku!"
  end
end
```

```
end
```

```
gimme { print "Powiedz jeszcze raz 'cześć'. " } #=> Powiedz  
↪jeszcze raz 'cześć'. Powiedz jeszcze raz 'cześć'.
```

Powinno się pamiętać również o tym, że po wykonaniu instrukcji `yield` sterowanie powraca do kolejnej instrukcji znajdującej się bezpośrednio po `yield`.

Procedury

Ruby pozwala na przechowywanie procedur (inaczej *procs*) jako obiektów, w całości wraz z ich kontekstami. Można to zrobić na kilka sposobów. Można utworzyć procedurę za pomocą wywołania metody `new` klasy `Proc` bądź też przez wywołanie metod `lambda` lub `proc` z modułu `Kernel`. Lepiej jest wywołać metody `lambda` lub `proc` niż `Proc.new`, ponieważ dwie pierwsze sprawdzają parametry. Można rozważyć poniższy przykład:

```
count = Proc.new { [1,2,3,4,5].each do |i| print i end; puts }  
your_proc = lambda { puts "Lurch: 'Wzywała mnie Pani?'" }  
my_proc = proc { puts "Morticia: 'Kto dzwonił do drzwi,  
↪Lurch?'" }
```

```
# Jakie rodzaje obiektów właśnie utworzono?
```

```
puts count.class, your_proc.class, my_proc.class
```

```
# Wywołanie wszystkich procedur
```

```
count.call #=> 12345
```

```
your_proc.call #=> Lurch: 'Wzywała mnie Pani?'
```

```
my_proc.call #=> Morticia: 'Kto dzwonił do drzwi, Lurch?'
```

Można dokonać konwersji bloku przekazanego jako argument metody na obiekt `Proc` dzięki poprzedzeniu nazwy argumentu znakiem `&`, jak poniżej:

```
def return_block  
  yield  
end
```

```
def return_proc( &proc )  
  yield
```

```
end
```

```
return_block { puts "Mam blok!" }  
return_proc { puts "Mam blok, przekonwertuj na procedurę!" }
```

Metoda `return_block` nie ma żadnych argumentów. Jedyne, co zawiera, to instrukcja `yield`. Celem instrukcji `yield` ponownie jest wykonanie bloku, kiedy zostanie on przekazany do metody. Kolejna metoda, `return_proc`, ma jeden argument — `&proc`. Kiedy argument metody poprzedzony jest znakiem `&`, metoda ta przyjmuje blok po jego przekazaniu i konwertuje go na obiekt `Proc`. Dzięki instrukcji `yield` znajdującej się w ciele metody wykonuje ona blok z procedurą bez konieczności angażowania metody `call` obiektu `Proc`.

Instrukcje warunkowe

Instrukcja warunkowa sprawdza, czy jakaś instrukcja zwraca `true`, czy też `false`, i wykonuje pewien kod w oparciu o wynik tego testu. Zarówno `true`, jak i `false` są pseudozmiennymi — nie można do nich przypisać wartości. Pierwsza jest obiektem klasy `TrueClass`, a druga — klasy `FalseClass`.

Instrukcja if

Instrukcje te rozpoczynają się od `if` i kończą się `end`:

```
if x == y then puts "x równa się y" end  
if x != y: puts "x nie jest równe y" end  
if x > y  
  puts "x jest większe od y"  
end
```

Separator `then` (oraz jego synonim `:`) są opcjonalne, o ile instrukcja nie jest zapisana w jednym wierszu.