helion.pl
KSIĘGARNIA INTERNETOWA · WYDAWNICTWO

IDŹ DO
PRZYKŁADOWY ROZDZIAŁ
SPIS TREŚCI

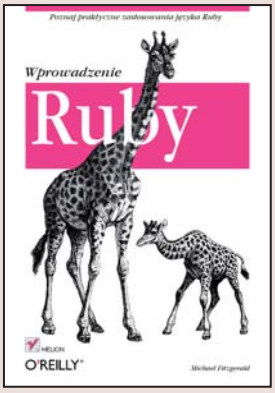
KATALOG KSIĄŻEK
KATALOG ONLINE
ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK
DODAJ DO KOSZYKA

CENNIK I INFORMACJE
ZAMÓW INFORMACJE O NOWOŚCIACH
ZAMÓW CENNIK

CZYTELNIA
FRAGMENTY KSIĄŻEK ONLINE

Ruby. Wprowadzenie



Autor: Michael Fitzgerald
Tłumaczenie: Adam Jarczyk
ISBN: 978-83-246-1229-1
Tytuł oryginału: Learning Ruby
Format: B5, stron: 240

Poznaj praktyczne zastosowania języka Ruby

- Podstawowe konstrukcje języka i zasady programowania obiektowego
- Operacje na tekstach, liczbach i plikach
- Framework Ruby on Rails

Ruby – obiektowy język programowania, względnie nowy, bo opracowany na początku lat 90. ubiegłego wieku, zdobywa coraz większą popularność. W zakresie tworzenia aplikacji internetowych staje się poważną konkurencją dla Perla, PHP i Javy. Jest niezwykle elastyczny, posiada prostą składnię i spore możliwości, a tworzony w nim kod jest bardzo zwarty. Za pomocą Ruby można pisać zarówno proste skrypty administracyjne, jak i rozbudowane aplikacje internetowe. W budowaniu tych ostatnich niezwykle pomocny jest framework Ruby on Rails, dzięki któremu proces tworzenia aplikacji przebiega błyskawicznie.

Książka „Ruby. Wprowadzenie" to podręcznik dla tych, którzy chcą poznać możliwości języka bez konieczności studiowania zawiłych opisów teoretycznych. Przedstawia Ruby na praktycznych przykładach, pokazując jego zastosowania w typowych zadaniach, z jakimi spotyka się na co dzień programista aplikacji sieciowych. Czytając tę książkę, poznasz elementy języka Ruby i nauczysz się programować obiektowo. Dowiesz się, w jaki sposób przetwarzać dane liczbowe, teksty i tablice, pliki i katalogi oraz dokumenty XML. Przeczytasz także o środowisku Ruby on Rails.

- Instalacja języka Ruby w różnych systemach operacyjnych
- Instrukcje i operatory
- Przetwarzanie tekstów i operacje matematyczne
- Operacje na systemie plików
- Korzystanie z plików XML
- Programowanie obiektowe
- Wprowadzenie do Ruby on Raili

Wejdź do świata Ruby, a pokochasz jego możliwości!

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl

Spis treści

ROZDZIAŁ 2.

Krótka wycieczka po języku Ruby

Niniejszy rozdział przedstawia, bez zbytniego zagłębiania się w szczegóły, podstawy języka Ruby: klasy i moduły, w tym klasę `Object` i moduł `Kernel`, słowa zastrzeżone (inaczej słowa kluczowe), komentarze, zmienne, metody i tak dalej. Większość z tych zagadnień będzie omówiona bardziej szczegółowo w innych rozdziałach. Niektóre tematy zasługują na całe rozdziały, inne tylko na podrozdziały (zawarte w rozdziale 10.). Za każdym razem powiem, gdzie znajdują się dodatkowe informacje na dany temat. W niniejszym rozdziale zostały zawarte najbardziej szczegółowe opisy metod i bloków.

Ruby jest językiem obiektowym

Matz, twórca języka Ruby, już w szkole średniej marzył o stworzeniu własnego języka programowania. Chciał utworzyć język skryptowy, który zarazem byłby zorientowany obiektowo.

Ruby wykracza poza proste pisanie skryptów, mimo że programy w nim mogą sprawiać wrażenie zwykłych skryptów powłoki. Nie jest to zaledwie język proceduralny, choć może być jako taki wykorzystany.

Język Ruby zawiera **klasy**. Klasy zawierają dane — w postaci stałych i zmiennych — oraz metody, które są krótkimi fragmentami kodu, pomagającymi wykonywać operacje na danych. Klasy mogą dziedziczyć po sobie informacje, lecz tylko z jednej klasy na raz. Pozwala to ponownie wykorzystywać kod — co oznacza mniej czasu zmarnowanego na naprawianie kodu i usuwanie z niego błędów — oraz mieszać ze sobą kod poprzez dziedziczenie.

Klasa jest czymś w rodzaju „schematu"; za pomocą metody `new` schemat ten można przypisać do zmiennej lub wykonać jego kopię, która przez to staje się **obiektem**. W języku Ruby obiektem jest prawie wszystko; w istocie to, co Ruby może skojarzyć z nazwą zmiennej, jest zawsze obiektem.

O klasach można powiedzieć o wiele więcej; Czytelnik znajdzie mnóstwo dodatkowych informacji na ich temat w rozdziale 9. Na razie wystarczą nam podstawy. Listing 2.1 przedstawia program w Ruby (*friendly.rb*) zawierający dwie klasy: `Hello` i `Goodbye`. Program ten jest zawarty w archiwum kodu Ruby towarzyszącym niniejszej książce (dostępnym pod adresem *ftp://ftp.helion.pl/przyklady/rubwpr.zip*). Proponuję uruchomić program w powłoce lub wierszu poleceń w katalogu, do którego zostało rozpakowane archiwum. W przypadkach gdy przykład kodu nie znajduje się w pliku, można wpisać go w irb. Zachęcam Czytelnika do wypróbowywania kodu tak często, jak to możliwe.

Listing 2.1. friendly.rb

```
class Hello
  def howdy
    greeting = "Witaj, Matz!"
    puts greeting
  end
end

class Goodbye < Hello
  def solong
    farewell = "Do widzenia, Matz."
    puts farewell
  end
end

friendly = Goodbye.new
friendly.howdy
friendly.solong
```

Po uruchomieniu tego programu zostaną wyświetlone komunikaty:

```
$ friendly.rb
Witaj, Matz!
Do widzenia, Matz.
```

Doświadczeni programiści zapewne bez podpowiedzi zorientują się, jak działa kod z listingu 2.1. Czytelnicy tacy mogą przejść do następnego punktu (lub prosto do rozdziału 9., aby poznać całą kwestię klas w języku Ruby).

Klasa `Hello` definiuje metodę `howdy`. Metoda ta wyświetla zawartość łańcucha zawartego w zmiennej `greeting` (`Witaj, Matz!`). Klasa `Goodbye` podobnie zawiera definicję metody `solong`, która wyświetla łańcuch przypisany do zmiennej `farewell` (`Do widzenia, Matz.`). Klasa `Goodbye` dziedziczy też zawartość klasy `Hello`; do tego służy tutaj operator <. Oznacza to, że w klasie `Goodbye` nie trzeba ponownie definiować metody `howdy`. Została po prostu odziedziczona.

`friendly` jest obiektem, egzemplarzem klasy `Goodbye`. Metoda `new` wywoływana z `Goodbye` pochodzi z klasy `Object` i tworzy nowy egzemplarz `friendly` (więcej na temat klasy `Object` w następnym punkcie). Mogliśmy użyć obiektu `friendly` do wywołania zarówno metody `howdy`, jak i `solong`, ponieważ obie są dostępne w tym obiekcie. Metoda `solong` została zdefiniowana w klasie `Goodbye`, a metoda `howdy` odziedziczona z `Hello`.

To wszystko, co na razie chciałem powiedzieć na ten temat. Informacje o klasach znajdą się w różnych miejscach następnych rozdziałów. W rozdziale 9. omówię klasy bardziej dokładnie.

Klasa Object i moduł Kernel

`Object` jest podstawową klasą języka Ruby, nadrzędną względem wszystkich pozostałych klas tego języka, i zawsze w sposób magiczny pojawia się, gdy uruchamiamy program w Ruby. Nie musimy niczego robić, by w innych klasach uzyskać dostęp do jej funkcjonalności. Jest zawsze dla nas dostępna.

Klasa `Object` zawiera bogatą funkcjonalność w postaci metod i stałych, którą wszystkie programy w Ruby dziedziczą automatycznie. W niniejszym punkcie przedstawię część tej funkcjonalności.

Klasa — analogia do wiadra

Czytelnicy, którzy nie wiedzą, na czym polega obiektowość języka programowania, mogą posłużyć się prostą analogią. Wyobraźmy sobie klasę — podstawowy składnik języka obiektowego — jako wiadro. W wiadrze znajduje się woda i jeden lub więcej czerpaków. Woda to odpowiednik właściwości (danych lub informacji) mieszczących się w klasie, a czerpaki są narzędziami (metodami) do manipulowania wodą (danymi). Podstawowym narzędziem używanym z klasami jest metoda — porcja kodu, której możemy nadać nazwę i wielokrotnie ją wykorzystywać. Metoda przypomina czerpak, który zanurzamy w wiadrze i wybieramy wodę lub nalewamy ją. Możemy wiadro wykorzystać ponownie, wylać starą wodę, nalać świeżej, a nawet włożyć jedno wiadro do drugiego. Tak, bez zagłębiania się w żargon techniczny, wyglądają podstawy programowania obiektowego. Sporą porcją żargonu poczęstuje Czytelnika rozdział 9.

`Object` udostępnia metody takie, jak `==` i `eql?`, `class`, `inspect`, `object_id` oraz `to_s`. Zostaną one dokładniej opisane w następnych rozdziałach. Więcej informacji o wszystkich metodach klasy `Object` można znaleźć pod adresem *http://www.ruby-doc.org/core/classes/Object.html*.

`Kernel` jest **modułem** języka Ruby. Moduł przypomina klasę, lecz nie można utworzyć jego egzemplarza, tak jak w przypadku klasy. Jeśli jednak dołączymy moduł do klasy lub wmieszamy go w nią, w obrębie tej klasy uzyskamy dostęp do wszystkich jego metod. Możemy używać metod z dołączonego modułu bez potrzeby ich implementowania.

Klasa `Object` zawiera moduł `Kernel`. Ponieważ zawsze mamy w programach Ruby dostęp do klasy `Object`, oznacza to, że zawsze uzyskujemy dostęp również do wszystkich metod modułu `Kernel`. Kilka z tych metod widzieliśmy już w działaniu, na przykład `print` i `puts`. Do najczęściej używanych metod modułu `Kernel` należą `eval`, `exit`, `gets`, `loop`, `require`, `sleep` i `sprintf`. W dalszych rozdziałach skorzystamy z większości tych metod.

Nazwy metody modułu `Kernel` nie trzeba poprzedzać nazwą obiektu lub odbiornika. Wystarczy wywołać metodę w dowolnym miejscu programu. Więcej informacji o module `Kernel` można znaleźć pod adresem *http://www.ruby-doc.org/core/classes/Kernel.html*.

Słowa zastrzeżone języka Ruby

Każdy język programowania ma własną listę **słów zastrzeżonych** (inaczej słów kluczowych), które zarezerwowane są na potrzeby działania języka. Słowami takimi są instrukcje w programach, a jak bez instrukcji mielibyśmy poinformować komputer, co ma robić?

Tabela 2.1 przedstawia listę słów zastrzeżonych języka Ruby wraz z krótkim opisem przeznaczenia każdego z nich.

Tabela 2.1. Słowa zastrzeżone języka Ruby

Słowo zastrzeżone	Opis
`BEGIN`	Kod zamknięty w nawiasy klamrowe `{}` wykonywany przed samym programem.
`END`	Kod zamknięty w nawiasy klamrowe `{}` wykonywany po zakończeniu działania programu.
`alias`	Tworzy alias dla istniejącej metody, operatora lub zmiennej globalnej.
`and`	Operator logiczny; taki sam jak `&&`, lecz o niższym priorytecie (analogicznie jak `or`).
`begin`	Zaczyna blok kodu (grupę instrukcji) zakończony słowem kluczowym `end`.
`break`	Wychodzi z pętli `while` lub `until` albo z metody wewnątrz bloku.
`case`	Porównuje wyrażenie z warunkiem `when`; instrukcja zakończona przez `end` (zobacz `when`).
`class`	Definiuje klasę; definicja jest kończona przez `end`.
`def`	Definiuje metodę; definicja jest kończona przez `end`.
`defined?`	Operator specjalny sprawdzający, czy zmienna, metoda, metoda klasy bazowej lub blok istnieje.
`do`	Zaczyna blok i wykonuje jego zawartość, kończy się na `end`.
`else`	Wykonuje następujący po nim kod, jeśli poprzedni warunek (`if`, `elsif`, `unless` albo `when`) nie jest prawdziwy.
`elsif`	Wykonuje następujący po nim kod, jeśli poprzedni warunek (`if` albo `elsif`) nie jest prawdziwy.
`end`	Kończy blok kodu rozpoczęty przez `begin`, `def`, `do`, `if` itp.
`ensure`	Zawsze wykonuje kod po zakończeniu bloku; należy użyć po ostatnim `rescue`.
`false`	Wartość logiczna (boolowska) „fałsz”; egzemplarz klasy `FalseClass` (zobacz `true`).
`for`	Rozpoczyna pętlę `for`; używane ze słowem kluczowym `in`.
`if`	Wykonuje blok kodu, jeśli warunek jest prawdziwy. Blok jest zakończony przez `end` (porównaj z `unless`, `until`).
`in`	Używane w pętli `for` (zobacz `for`).
`module`	Definiuje moduł; definicja jest kończona przez `end`.
`next`	Wykonuje skok przed instrukcją warunkową pętli (porównaj z `redo`).
`nil`	Wartość pusta, niezainicjowana zmienna, niepoprawna, lecz nie to samo co zero. Obiekt z klasy `NilClass`.
`not`	Operator logiczny, taki sam jak `!`.
`or`	Operator logiczny; taki sam jak `||`, lecz o niższym priorytecie (analogicznie jak `and`).
`redo`	Skok po instrukcji warunkowej pętli (porównaj z `next`).
`rescue`	Ewaluuje wyrażenie po pojawieniu się wyjątku; używane przed `ensure`.
`retry`	Na zewnątrz `rescue` powtarza wywołanie metody; wewnątrz `rescue` wykonuje skok na początek bloku (`begin`).
`return`	Zwraca wartość z metody lub bloku. Można pominąć.
`self`	Obiekt bieżący (wywołane przez metodę).
`super`	Wywołuje metodę o tej samej nazwie w superklasie (klasie nadrzędnej względem bieżącej).
`then`	Kontynuacja instrukcji `if`, `unless` lub `when`. Można pominąć.
`true`	Wartość logiczna (boolowska) „prawda”; egzemplarz klasy `TrueClass` (zobacz `false`).
`undef`	Usuwa definicję metody w bieżącej klasie.
`unless`	Wykonuje blok kodu, jeśli instrukcja warunkowa zwróci `false` (porównaj z `if`, `until`).
`until`	Wykonuje blok kodu, gdy instrukcja warunkowa zwróci `false` (porównaj z `if`, `unless`).
`when`	Rozpoczyna klauzulę (jedną lub więcej) po `case`.

Tabela 2.1. Słowa zastrzeżone języka Ruby — ciąg dalszy

Słowo zastrzeżone	Opis
`while`	Wykonuje blok kodu, jeśli instrukcja warunkowa zwróci `true`.
`yield`	Wykonuje blok przekazany do metody.
`__FILE__`	Nazwa bieżącego pliku źródłowego.
`__LINE__`	Numer bieżącego wiersza w bieżącym pliku źródłowym.

Komentarze

Komentarz ukrywa wiersze skryptu przed interpreterem języka Ruby, tak że zostają odrzucone (zignorowane). Umożliwia to programistom (czyli nam) wstawianie do programów wszelkiego rodzaju informacji, które pozwolą innym użytkownikom zorientować się, o co chodzi. W Ruby stosowane są dwa podstawowe style komentarzy. Symbol # (hash) może znajdować się na początku wiersza:

```
# Jestem tylko komentarzem. Nie zwracaj na mnie uwagi.
```

albo po instrukcji lub wyrażeniu, w tym samym wierszu:

```
name = "Floydee Wallup" # też mi nazwisko...
```

Komentarz może zajmować kilka wierszy pod rząd:

```
# To jest komentarz.
# To też jest komentarz.
# I to też jest komentarz.
# Nie będę się powtarzać.
```

Oto inna forma: blokowy komentarz, który ukrywa przed interpreterem kilka wierszy pomiędzy słowami kluczowymi `=begin` i `=end`:

```
=begin
To jest komentarz.
To też jest komentarz.
I to też jest komentarz.
Nie będę się powtarzać.
=end
```

W ten sposób można zakomentować jeden wiersz lub dowolną ich liczbę.

Zmienne

Zmienna jest identyfikatorem (nazwą), któremu można przypisać wartość. Podczas działania programu wartość ma lub będzie mieć określony typ. W poniższym przykładzie zmiennej `x` zostaje za pomocą znaku równości przypisana wartość 100:

```
x = 100
```

Teraz zmienna lokalna `x` zawiera wartość 100. Ale jakiego typu? Dla mnie wygląda na liczbę całkowitą, a dla Czytelnika? A jak zinterpretuje ją Ruby?

Wiele współczesnych języków programowania, na przykład C++ i Java, stosuje statyczną kontrolę typu. Oznacza to, że zmiennej w chwili jej deklarowania przypisuje się określony typ, a ponieważ w językach tych kontrola typów jest ścisła, zmienna zachowuje typ, dopóki nie zostanie jej przypisany inny (o ile jest to w ogóle możliwe).

Na przykład, w języku Java zmienne deklaruje się z podaniem typu (`int`) po lewej stronie:

```
int months = 12;
int year = 2007;
```

W języku Ruby nie ma deklaracji typów. Wartości są po prostu przypisywane do zmiennych:

```
months = 12
year = 2007
```

Jeśli ktoś sobie tego życzy, może zakończyć wiersz średnikiem, lecz znak końca wiersza w zupełności wystarczy.

Wartości w zmiennych `x`, `months` i `year` są ewidentnie liczbami całkowitymi (*integer*), lecz nie musimy wskazywać typu, ponieważ Ruby robi to za nas automatycznie. Nosi to nazwę **dynamicznej kontroli typów** lub potocznie ***duck typing***.

Mechanizm działa tak: jeśli zauważymy ptaka wodnego, który chodzi jak kaczka (ang. *duck*), kwacze jak kaczka, fruwa jak kaczka i pływa jak kaczka, to, na honor, zapewne mamy do czynienia z kaczką. Ruby podobnie sprawdza wartość przypisaną do zmiennej — jeśli ta wartość chodzi, kwacze, fruwa i pływa jak liczba całkowita, Ruby przyjmuje, że może grać rolę liczby całkowitej.

Zobaczmy, czy Ruby może uznać wartość `x` za liczbę całkowitą, za pomocą metody `kind_of?` (jest to metoda z klasy `Object`).

```
x.kind_of? Integer # => true
```

Ależ oczywiście, wartość zmiennej `x` zachowuje się jak liczba całkowita! W istocie jest egzemplarzem klasy `Fixnum`, która dziedziczy klasę `Integer`.

```
x.class # => Fixnum
```

Zmieńmy teraz wartość `x` z całkowitej na zmiennoprzecinkową za pomocą metody `to_f` z klasy `Fixnum` (jest też dziedziczona przez inne klasy):

```
x.to_f # => 100.0
```

Jak wspomniałem w rozdziale 1., zapis => w przykładach kodu pojawia się tu zawsze po znaku komentarza (#). Tekst następujący po => jest wynikiem, którego możemy się spodziewać z wiersza lub bloku kodu albo całego programu.

Zmienne lokalne

Nazwałem wcześniej `x` **zmienną lokalną**. Co to znaczy? Oznacza to, że zmienna ma zasięg (kontekst) lokalny. Na przykład, gdy zmienna lokalna zostaje zdefiniowana **wewnątrz** metody lub pętli, ma zasięg w obrębie tej metody lub pętli. Na zewnątrz nie jest do niczego przydatna.

Nazwy zmiennych lokalnych muszą zaczynać się od małej litery lub znaku podkreślenia (_), na przykład `alpha` lub `_beta`. Zmienne lokalne w języku Ruby możemy też poznać po tym, że ich nazwy nie są poprzedzane znakami specjalnymi, z wyjątkiem podkreślenia. Istnieją też inne typy zmiennych, z łatwością identyfikowane przez pierwszy znak nazwy. Należą do nich zmienne globalne, zmienne instancji i zmienne klasy.

Zmienne instancji

Zmienna instancji jest zmienną, do której odwołujemy się przez wystąpienie (egzemplarz) klasy, więc należy do określonego obiektu. Nazwy takich zmiennych poprzedza znak `@`:

```
@hello = hello
```

Dostęp do zmiennej instancji spoza jej własnej klasy jest możliwy tylko przez metody akcesora. Więcej informacji o tych metodach zawiera rozdział 9.

Zmienne klasy

Zmienna klasy jest współużytkowana przez wszystkie wystąpienia klasy. Dla danej klasy istnieje tylko jedna kopia zmiennej klasy. W języku Ruby nazwy zmiennych tego typu poprzedzają dwa znaki `@` (`@@`). Przed użyciem zmiennej klasy należy ją zainicjalizować (przypisać wartość):

```
@@times = 0
```

Zmienne klasy zobaczymy w akcji w rozdziale 9.

Zmienne globalne

Zmienne globalne są dostępne globalnie w całym programie, w każdej jego strukturze. Ich zasięgiem jest cały program. Nazwę zmiennej globalnej poprzedza symbol dolara (`$`):

```
$amount = "0.00"
```

Trudno jest śledzić wykorzystanie zmiennych globalnych. Lepiej wyjdziemy na projektowaniu kodu tak, by wykorzystywał zmienne klasy i stałe. Matz twierdzi, cytuję, że zmienne globalne „są paskudne, więc nie korzystaj z nich". Sądzę, że to dobra rada (lepiej używać metod singleton; zobacz rozdział 9.).

Stałe

Stała mieści niezmienną wartość przez cały czas działania programu w Ruby. Stałe są zmiennymi, których nazwy zaczynają się od dużej litery lub są zapisane samymi dużymi literami. Oto definicja stałej o nazwie `Matz`:

```
Matz = "Yukishiro Matsumoto"
puts Matz # => Yukishiro Matsumoto
```

Jeśli stała została zdefiniowana wewnątrz klasy lub modułu, będzie dostępna w tej klasie lub module; jeśli zdefiniujemy ją na zewnątrz klas i modułów, będzie dostępna globalnie. W przeciwieństwie do innych języków w Ruby stałe są zmienialne (*mutable*) — inaczej mówiąc, można modyfikować ich wartości.

Przypisanie równoległe

Podoba mi się w języku Ruby możliwość przypisywania równoległego (jak w językach Perl, Python i JavaScript 1.7). Co to takiego? Jest to sposób przypisywania w jednej instrukcji kilku zmiennych, w jednym wierszu. Często przypisujemy po jednej zmiennej na wiersz:

```
x = 100
y = 200
z = 500
```

W przypisaniu równoległym możemy uzyskać ten sam efekt, oddzielając od siebie nazwy zmiennych, a następnie wartości przecinkami:

```
x, y, z = 100, 200, 500
```

Możliwe jest nawet przypisanie wartości różnych typów, na przykład łańcucha, wartości zmiennoprzecinkowej i całkowitej:

```
a, b, c = "cash", 1.99, 100
```

Przypisanie równoległe jest wygodne. To bardzo typowe dla języka Ruby.

Łańcuchy

Łańcuch jest ciągiem liter, cyfr i innych znaków. W języku Ruby istnieje kilka metod tworzenia łańcuchów, lecz najprostszą chyba jest zapisanie tekstu w cudzysłowach (mogą być pojedyncze lub podwójne). Weźmy cytat z książki *Walden* Henry Davida Thoreau:

```
thoreau = "Dobroć jest jedyną inwestycją, która nigdy nie zawodzi."[1]
```

Metody klasy `String` pozwalają uzyskać dostęp do łańcucha `thoreau` i manipulować nim. Możemy, na przykład, pobrać fragment łańcucha za pomocą metody `[]`, używając **zakresu**. Weźmy znaki z pozycji od 33. do 37.:

```
thoreau[33..37] # => "nigdy"
```

Albo, zaczynając od końca łańcucha i używając liczb ujemnych, weźmy znaki od przedostatniego do ósmego od końca:

```
thoreau[-8..-2] # => "zawodzi"
```

Możemy przejść iteracyjnie przez wszystkie znaki łańcucha, używając bloku kodu, który pobierze kolejno każdy bajt (8 kolejnych bitów) łańcucha i wygeneruje z niego znak za pomocą metody `chr`, oddzielając od siebie kolejne znaki ukośnikami:

```
thoreau.each_byte do |c|
  print.chr, "/"
end
# => D/o/b/r/o/ć/ /j/e/s/t/ /j/e/d/y/n/ą/ /i/n/w/e/s/t/y/c/j/ą/,/ /k/t/ó/r/a/ /n/i/g/d/y/ /n/i/e/ /z/a/w/o/d/z/i/./
```

Niniejsza wskazówka przeznaczona jest dla czytelników, którzy chcą w programach używać nie tylko znaków ASCII; w przeciwnym razie może być nadmiarem informacji. Muszę przyznać, że założenie, iż znak jest równoważny bajtowi, jest dość staroświeckie. W szerszym kontekście (w Unicode) znak może być reprezentowany przez więcej niż jeden bajt. Na przykład, w kodowaniu UTF-8 każdy znak jest reprezentowany przez od jednego do czterech bajtów. Ruby domyślnie stosuje kodowanie znaków ASCII, lecz możemy to zmienić, ustawiając (w okolicach początku programu) w zmiennej `$KCODE` wartość `u` (dla UTF-8), `e` (dla EUC), `s` (dla SJIS), `a` (ASCII) lub `n` (`NONE`).

[1] Tłumaczenie własne — *przyp. tłum.*

Tak przy okazji, metoda `chr` konwertuje kod znaku (wygenerowany przez `each_byte`) na faktyczny znak. Powinienem też wspomnieć o metodzie odwrotnej — operatorze `?`, który zwraca kod podanego znaku. Zademonstruję to w irb:

```
irb(main):001.0> ?I
=> 73
irb(main):002.0> ?f
=> 102
irb(main):003.0> ?\   # nie widać znaku, ale tu jest spacja
=> 32
irb(main):004.0> ?m
=> 109
irb(main):005.0> ?a
=> 97
irb(main):006.0> ?n
=> 110
```

Nie będę się tu zbytnio zagłębiać w szczegóły; więcej informacji o łańcuchach zawiera rozdział 4.

Wyrażenia regularne

Wyrażenie regularne jest specjalnym ciągiem znaków, który dopasowuje łańcuch lub zbiór łańcuchów. Wyrażenia regularne (potocznie „regexp" od *regular expression*) często są wykorzystywane do znajdowania pasujących łańcuchów, aby później zrobić coś z nimi lub wykonać inną operację. Służą też do pobierania łańcuchów lub podłańcuchów do wykorzystania gdzieś indziej.

Wyrażenia regularne składają się z elementów (jednego lub więcej znaków), które instruują silnik wyrażeń regularnych, jakie łańcuchy ma wyszukiwać. Na wzorzec wyrażenia regularnego składa się kombinacja znaków specjalnych zawarta w ukośnikach (`//`). Oto kilka przykładów:

`^`
: Dopasowuje początek wiersza.

`$`
: Dopasowuje koniec wiersza.

`\w`
: Dopasowuje słowo (dowolną kombinację liter i cyfr).

`[...]`
: Dopasowuje dowolny znak w nawiasach prostokątnych.

`[^...]`
: Dopasowuje dowolny znak, który nie znajduje się w nawiasach prostokątnych.

`*`
: Dopasowuje zero lub więcej wystąpień poprzedniego wyrażenia regularnego.

`+`
: Dopasowuje jedno lub więcej wystąpień poprzedniego wyrażenia regularnego.

`?`
: Dopasowuje zero lub jedno wystąpienie poprzedniego wyrażenia regularnego.

Oto przykład wyrażenia regularnego dopasowującego łańcuchy za pomocą metody `scan` z klasy `String`:

```
hamlet = "Znosić pociski zawistnego losu"
hamlet.scan(/\w+/) # => ["Znosić", "pociski", "zawistnego", "losu"]
```

Dopasowuje ono jedno lub więcej (+) słów (`\w`) i umieszcza znalezione wyniki w tablicy.

W szczegóły wyrażeń regularnych zagłębię się w rozdziale 4. Znajduje się w nim tabela wszystkich wzorców wyrażeń regularnych rozpoznawanych przez język Ruby. Czytelników poważnie zainteresowanych tym tematem odsyłam do książki Jeffreya E. F. Friedla *Wyrażenia regularne* (Helion, Gliwice 2001).

Liczby i operatory

W większości obiektowych języków programowania liczby są uznawane za podstawowe elementy składowe nazywane **prymitywami** lub **elementami pierwotnymi**. Nie są wprost powiązane z żadną klasą; po prostu *są*. W języku Ruby jest inaczej: nawet liczby są egzemplarzami klas.

Dodatnia liczba całkowita 1001, na przykład, jest egzemplarzem klasy `Fixnum`, która jest klasą potomną klasy `Integer`, a ta z kolei potomną klasy `Numeric`. Zmiennoprzecinkowa liczba 1001.0 jest egzemplarzem klasy `Float`, która również jest klasą potomną `Numeric` (rysunek 5.1 przedstawia relacje pomiędzy tymi klasami).

Wraz z liczbami pojawiają się operacje, które na tych liczbach można wykonywać. Możemy je, na przykład, dodawać, dzielić, mnożyć, podnosić do potęgi czy też zwracać resztę z dzielenia (modulo).

Doskonałym narzędziem do zaznajomienia się z działaniami matematycznymi w Ruby jest irb. Wypróbujmy kilka z nich:

```
irb(main):001.0> 3 + 4 # dodawanie
=> 7
irb(main):002.0> 7 - 3 # odejmowanie
=> 4
irb(main):003.0> 3 * 4 # mnożenie
=> 12
irb(main):004.0> 12 / 4 # dzielenie
=> 3
irb(main):005.0> 12**2 # podniesienie do potęgi
=> 144
irb(main):006.0> 12 % 7 # modulo (reszta)
=> 5
```

Oto kilka operatorów przypisania języka Ruby w akcji:

```
irb(main):007.0> x = 12 # przypisanie
=> 12
irb(main):008.0> x += 1 # skrócony zapis przypisania z dodaniem
=> 13
irb(main):009.0> x -= 1 # skrócony zapis przypisania z odejmowaniem
=> 12
irb(main):010.0> x *= 2 # skrócony zapis przypisania z mnożeniem
=> 3
irb(main):011.0> x /= 2 # skrócony zapis przypisania z dzieleniem
=> 144
```

Ruby zawiera również moduł `Math`, udostępniający wszelkiego rodzaju funkcje matematyczne (w postaci metod klas), takie jak pierwiastek kwadratowy, cosinus, tangens i tak dalej. Oto przykład wywołania metody klasy `sqrt` (pierwiastek kwadratowy) z modułu `Math`:

```
irb(main):012.0> Math.sqrt(16)
=> 4.0
```

Poza tym Ruby udostępnia kilka specjalnych klas matematycznych, na przykład `Rational` do operacji na ułamkach. Więcej o liczbach i operatorach powiem w rozdziale 5. Tabela 5.1 przedstawia wszystkie operatory matematyczne w języku Ruby, łącznie z pierwszeństwem.

Instrukcje warunkowe

Podobnie jak każdy inny język programowania, Ruby zawiera **instrukcje warunkowe**, które sprawdzają, czy określone stwierdzenie jest prawdziwe, czy fałszywe. Na podstawie odpowiedzi jest następnie wykonywany blok kodu. Oto prosty przykład instrukcji `if`, w której sprawdzamy, czy zmienna ma wartość zero:

```
value = 0

if value.zero? then
  puts "Zmienna value ma wartość 0. Domyśliłeś się tego?"
end
```

Metoda `zero?` zwraca `true`, jeśli wartość zmiennej `value` wynosi zero. Tak też jest w naszym przypadku, więc następna instrukcja zostaje wykonana (wraz z wszelkimi innymi instrukcjami w bloku kodu pomiędzy `if` a `end`). Zgodnie z konwencją języka Ruby każda metoda, której nazwa zakończona jest znakiem zapytania, zwraca wartość boolowską — `true` albo `false`. Konwencja ta nie jest jednak wymuszana.

Do innych instrukcji warunkowych należą na przykład znane nam `case` i `while` oraz mniej znane, jak na przykład `until` i `unless`. Rozdział 3. opisuje wszystkie instrukcje warunkowe dostępne w języku Ruby.

Tablice i tablice asocjacyjne

Tablica jest uporządkowaną sekwencją indeksowanych wartości, w której indeks zaczyna się od zera. Stanowi jedną z najczęściej spotykanych w informatyce struktur danych. W języku Ruby tablica może wyglądać tak:

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

Nasza tablica ma nazwę `pacific`. Zawiera trzy łańcuchy — nazwy trzech stanów składających się na zachodnie wybrzeże USA. Łańcuchy te są **elementami** tablicy. Elementami tablicy mogą być, oczywiście, dane dowolnego typu z języka Ruby, nie jedynie łańcuchy. Powyższy przykład jest tylko jednym ze sposobów definiowania tablicy. Istnieje wiele innych, które poznamy w rozdziale 6.

Aby uzyskać dostęp do wybranego elementu, można podać w metodzie jego indeks. Na przykład, by pobrać pierwszy element, którego indeks ma wartość zero, możemy użyć metody `[]`:

```
pacific[0] # => "Waszyngton"
```

Wywołanie tej metody zwraca wartość elementu zerowego — łańcuch `Waszyngton`. Więcej informacji o tablicach w języku Ruby zawiera rozdział 6.

Tablica asocjacyjna (ang. *hash*), która przypisuje klucze do wartości, również jest powszechnie spotykaną strukturą danych. W przeciwieństwie do zwykłej tablicy, w której indeksami są dodatnie liczby całkowite, w tablicy asocjacyjnej możemy dowolnie wybrać klucze służące do indeksowania. Wygląda to tak:

```
pacific = { "WA" => "Waszyngton", "OR" => "Oregon", "CA" => "Kalifornia" }
```

Definicja tablicy asocjacyjnej jest zamknięta w nawiasy klamrowe, natomiast zwykłej w nawiasy prostokątne. Poza tym każda wartość jest kojarzona z kluczem za pomocą operatora =>. Jedną z metod dostępu do wartości w tablicy asocjacyjnej jest podanie klucza. Aby pobrać z naszej tablicy asocjacyjnej wartość Oregon, możemy użyć metody `[]` z klasy `Hash`:

```
pacific["OR"] # => "Oregon"
```

Podając klucz `OR`, otrzymaliśmy wartość `Oregon`. Klucze i wartości mogą być dowolnego typu, nie tylko łańcuchami. Więcej informacji o tablicach asocjacyjnych zawiera rozdział 7.

Metody

Metody pozwalają zgrupować kod (instrukcje i wyrażenia) w jednym miejscu, tak że można go później wygodnie wykorzystać — jeśli będzie trzeba, wielokrotnie. Możemy definiować metody wykonujące najróżniejsze czynności. W istocie większość operatorów matematycznych w języku Ruby to metody.

Poniższy opis metod jest najbardziej zwięzły ze wszystkich zawartych w niniejszej książce, więc Czytelnik być może zechce powracać do niego w trakcie dalszej lektury.

Oto prosta definicja metody o nazwie `hello`, stworzona z użyciem słów kluczowych `def` i `end`:

```
def hello
   puts "Witaj, Matz!"
end
```

Ta metoda po prostu wyświetla tekst za pomocą instrukcji `puts`. Z drugiej strony, możemy usunąć definicję metody instrukcją `undef`:

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

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

Możemy też definiować metody mające argumenty, jak w poniższym przykładzie:

```
def repeat ( word, times )
   puts word * times
end
repeat("Witaj! ", 3) # => Witaj! Witaj! Witaj!
repeat("Żegnaj! ", 4) # => Żegnaj! Żegnaj! Żegnaj! Żegnaj!
```

Nasza metoda `repeat` przyjmuje dwa argumenty: `word` i `times`. Metodę mającą zdefiniowane argumenty możemy wywoływać z nawiasami lub bez przed i po argumentach. Można nawet definiować argumenty metody bez nawiasów, lecz zwykle tak nie robię.

Dzięki temu, że nawiasy nie są wymagane, możemy zapisywać normalnie wyglądające równania matematyczne, korzystając z metod operatorów, na przykład +. Każdy z trzech poniższych wierszy jest w rzeczywistości poprawnym wywołaniem metody + klasy `Fixnum`:

```
10 + 2 # => 12
10. + 2 # => 12
(10). + (2) # => 12
```

Wartości zwracane

Metody zwrają wartości. W innych językach programowania możemy jawnie otrzymać zwracaną wartość za pomocą instrukcji `return`. W języku Ruby zwracana jest ostatnia wartość z metody, z użyciem instrukcji `return` lub bez. Taki jest styl Ruby. Oto, jak możemy to sprawdzić w irb:

1. Na początek zdefiniujemy metodę `matz` zawierającą jedynie łańcuch:

```
irb(main):001:0> def matz
irb(main):002:1> "Witaj, Matz!"
irb(main):003:1> end
=> nil
```

2. Po wywołaniu metody `matz` zobaczymy jej wyjście. Jest to dostępne w irb, lecz nie będzie widoczne w przypadku uruchomienia normalnego programu z wiersza zachęty powłoki. Aby wyświetlić wyjście, w programie użylibyśmy instrukcji `puts`:

```
irb(main):004:0> matz
=> "Witaj, Matz!"
irb(main):005:0> puts matz
"Witaj, Matz!"
=> nil
```

3. Przypiszemy teraz metodę `matz` do zmiennej `output` i wyświetlimy tę zmienną:

```
irb(main):006:0> output = matz
=> "Witaj, Matz!"
irb(main):007:0> puts output
"Witaj, Matz!"
=> nil
```

4. Jeśli ktoś sobie tego zażyczy, może użyć jawnie instrukcji `return`. Zdefiniujemy ponownie metodę `matz`, tym razem dodając instrukcję `return`, i otrzymamy ten sam wynik:

```
irb(main):008:0> def matz
irb(main):009:1> return "Witaj, Matz!"
irb(main):010:1> end
=> nil
irb(main):011:0> matz
"Witaj, Matz!"
irb(main):012:0> puts matz
"Witaj, Matz!"
=> nil
irb(main):013:0> output = matz
=> "Witaj, Matz!"
irb(main):014:0> puts output
"Witaj, Matz!"
=> nil
```

Konwencje nazewnicze metod

W języku Ruby stosowane są konwencje dotyczące ostatniego znaku nazwy metody — konwencje powszechnie stosowane, lecz nie wymuszane przez język.

Gdy nazwa metody kończy się znakiem zapytania, na przykład `eql?`, metoda zwraca wartość boolowską — `true` albo `false`. Na przykład:

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

Gdy nazwa metody kończy się wykrzyknikiem, na przykład `delete!`, wskazuje to, że metoda jest „niszczycielska" — inaczej mówiąc, wprowadza zmiany w samym obiekcie, a nie w jego kopii. Modyfikuje sam obiekt. Proszę zwrócić uwagę na różnice wyników działania metod `delete` i `delete!` klasy `String`:

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

Gdy nazwa metody kończy się znakiem równości, na przykład `family_name=`, oznacza to, że metoda jest tzw. „setterem" — przeprowadza przypisanie lub ustawia zmienną, na przykład zmienną 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">
```

W języku Ruby dostępna jest bardziej praktyczna technika tworzenia metod getter/setter i akcesorów. Została ona opisana w rozdziale 9.

Argumenty domyślne

Metoda `repeat`, przedstawiona wcześniej, ma dwa argumenty. Możemy im przypisać wartości domyślne, używając znaku równości, a po nim wartości. Jeśli wywołamy metodę bez argumentów, automatycznie zostaną użyte wartości domyślne.

Zdefiniujmy `repeat` ponownie, używając wartości domyślnych: `Witaj!` dla `word` i 3 dla `times`. Wywołamy teraz metodę najpierw bez argumentów, a następnie z nimi.

```
def repeat ( word="Witaj! ", times=3 )
   puts word * times
end
repeat # => Witaj! Witaj! Witaj!
repeat("Żegnaj! ", 5 ) # => Żegnaj! Żegnaj! Żegnaj! Żegnaj! Żegnaj!
```

Gdy metodę wywołaliśmy bez argumentów, zostały użyte wartości domyślne; lecz gdy podaliśmy argumenty, wartości domyślne zostały odrzucone i zastąpione wartościami argumentów.

Zmienna liczba argumentów

Czasem nie wiemy, ile argumentów będzie miała metoda. Możemy pozwolić sobie na elastyczność, ponieważ Ruby umożliwia przekazanie do metody zmiennej liczby argumentów, poprzedzając po prostu argument symbolem *. Listing 2.2 przedstawia prosty program, który wykorzystuje tę metodę.

Listing 2.2. num_args.rb

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

puts num_args

puts num_args(1)

puts num_args( 100, 2.5, "three" )
```

W tym programie operator trójkowy (?:) posłużył do ustalenia, czy rzeczownik `argument` powinien być w liczbie pojedynczej, czy mnogiej (więcej o operatorach trójkowych w następnym rozdziale).

Gdy użyjemy tej składni dla różnej liczby argumentów, będą one zapisane w tablicy, co pozwoli zilustrować metoda `inspect`. Trzy wywołania `num_args` są poprzedzone instrukcją `puts`, aby wysłać wartość zwracaną metody na wyjście standardowe:

```
0 arguments ( [] )
1 argument ( [1] )
3 arguments ( [100, 2.5, "three"] )
```

Jednocześnie z argumentami zmiennymi możemy używać zwykłych. Cała sztuka polega na tym, że zmienna lista argumentów (zaczynająca się od *) zawsze powinna znajdować się na końcu listy. Listing 2.3 ilustruje metodę, która ma dwa zwykłe argumenty i miejsce na dodatkowe.

Listing 2.3. two_plus.rb

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

puts two_plus( 1, 2 )

puts two_plus( 1000, 3.5, 14.3 )

puts two_plus( 100, 2.5, "three", 70, 14.3 )
```

Oto wyjście programu (ilustruje tylko liczbę argumentów zmiennych, ignorując zwykłe):

```
0 variable arguments ( [] )
1 variable argument ( [14.3] )
3 variable arguments ( [100, 2.5, "three"] )
```

Proszę spróbować wywołać `two_plus` bez żadnych argumentów i sprawdzić, jaka będzie odpowiedź interpretera.

Aliasy metod

Język Ruby zawiera słowo kluczowe `alias`, które tworzy aliasy metod. Oznacza to, że tworzymy kopię metody pod nową nazwą, aczkolwiek wywołanie obu metod będzie kierowane do tego samego obiektu. Za pomocą `alias` (lub metody `alias_method` klasy `Module`) możemy uzyskać dostęp do metod, które zostały nadpisane.

Poniższy przykład z irb ilustruje, jak utworzyć alias metody `greet`:

```
irb(main):001:0> def greet
irb(main):002:1>   puts "Cześć, dziecinko!"
irb(main):003:1> end
=> nil
irb(main):004:0> alias baby greet # tworzymy alias greet pod nazwą baby
=> nil
irb(main):005:0> greet # wywołujemy metodę
Cześć, dziecinko!
=> nil
irb(main):006:0> baby # wywołujemy alias
Cześć, dziecinko!
=> nil
irb(main):007:0> greet.object_id # jaki jest identyfikator obiektu?
Cześć, dziecinko!
=> 4
irb(main):008:0> baby.object_id # wskazuje ten sam obiekt
Cześć, dziecinko!
=> 4
```

Bloki

Blok w języku Ruby jest czymś więcej niż fragmentem kodu (grupą instrukcji). W określonym kontekście blok ma specjalne znaczenie. Jak zobaczymy, tego typu blok jest zawsze wywoływany w połączeniu z metodą. W istocie określany jest terminem **funkcja nienazwana**.

W Ruby blok jest często (lecz nie zawsze) sposobem na pobranie wszystkich wartości ze struktury danych przez iterację na tej strukturze. Oznacza mniej więcej „daj mi wszystko, co tu masz, po jednym na raz”. Pokażę tu typowe zastosowanie bloku.

Przypomnijmy sobie tablicę `pacific`:

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

Możemy wywołać blok na tej tablicy, aby pobrać po jednym wszystkie elementy, posługując się metodą `each`. Oto jeden z możliwych sposobów:

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

Nazwa pomiędzy znakami | (|element|) może być dowolna. Blok wykorzystuje ją jako zmienną globalną do śledzenia elementów tablicy, a później do zrobienia czegoś z każdym elementem po kolei. W naszym przykładzie w bloku instrukcja puts wyświetla kolejno wszystkie elementy tablicy:

```
Waszyngton
Oregon
Kalifornia
```

Instrukcje do..end można zastąpić parą nawiasów klamrowych, co też powszechnie jest stosowane, by otrzymać bardziej zwięzły kod (tak przy okazji, nawiasy klamrowe mają wyższy priorytet niż do..end):

```
pacific.each { |e| puts e }
```

Metoda each jest dostępna w dziesiątkach klas, między innymi w Array, Hash i String. Ale proszę nie sugerować się tym zanadto. Iteracja przez struktury danych nie jest jedyną metodą wykorzystania bloków. Pozwolę sobie przytoczyć prosty przykład wykorzystujący słowo kluczowe Ruby yield.

Instrukcja yield

Na początek zdefiniujmy króciutką metodę gimme zawierającą jedynie instrukcję yield:

```
def gimme
   yield
end
```

Aby sprawdzić, co robi ta metoda, wywołamy ją bez niczego i zobaczymy efekt:

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

Otrzymaliśmy komunikat o błędzie, ponieważ zadaniem instrukcji yield jest wykonanie bloku kodu skojarzonego z metodą. Tego nam brakowało w wywołaniu gimme. Możemy uniknąć takiego błędu, posługując się metodą block_given? z modułu Kernel. Zdefiniujmy gimme, dodając instrukcję if:

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

if jest instrukcją warunkową. Jeśli z wywołaniem metody został podany blok, metoda block_given? zwróci true i yield wykona blok; w przeciwnym razie zostanie wykonany kod po else.

Wypróbujmy metodę jeszcze raz, z blokiem i bez.

```
gimme { print "Dzień dobry wszystkim."} # => Dzień dobry wszystkim.

gimme # => Nie mam bloku!
```

Gdy podajemy `gimme` blok, metoda wykonuje zawarty w nim kod, wyświetlając tekst Dzień dobry wszystkim; jeśli nie podamy bloku, `gimme` zwróci łańcuch `Nie mam bloku!`. Z ciekawości możemy zmienić definicję `gimme` tak, że będzie zawierać dwie instrukcje `yield`, a następnie wywołać z blokiem. Wykona blok dwa razy.

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

gimme { print "Dzień dobry."} # => Dzień dobry. Dzień dobry.
```

Należy pamiętać, że po wykonaniu instrukcji `yield` program wraca do następnej instrukcji bezpośrednio po `yield`. Aby to zilustrować, zdefiniujmy `gimme` jeszcze raz.

```
def gimme
   if block_given?
      yield
   else
      puts "Oh-oh. Brak bloku."
   end
   puts "Cała przyjemność po mojej stronie."
end

gimme { print "Dziękuję."} # => Dziękuję. Cała przyjemność po mojej stronie.
```

Jestem pewien, że te drobne przykłady kodu zademonstrowały, jak wszechstronnym narzędziem są bloki. Mogę sobie wyobrazić eksplozję kreatywności w głowie Czytelnika.

Aby w pełni poznać możliwości bloków, musimy powiedzieć coś o procedurach.

Bloki są domknięciami

Kto z czytelników wie, czym jest *domknięcie*? Jeśli ktoś wie, jestem pełen podziwu — zapewne ta wiedza świadczy o zdobytym dyplomie magistra informatyki. Tym, którzy nie wiedzą, a są ciekawi, już wyjaśniam. Domknięcie jest nienazwaną funkcją lub metodą. Przypomina metodę wewnątrz metody, odwołującą się do zmiennych z metodą zewnętrzną lub je współużytkującą. W języku Ruby domknięcie lub blok są zamknięte w nawiasy klamrowe (`{}`) lub słowa kluczowe `do..end` i ich działanie jest uzależnione od odpowiedniej metody (na przykład `each`).

Obiekty Proc

Ruby pozwala przechowywać **procedury** (potocznie *proc*) jako obiekty, razem z ich kontekstem. Dostępnych jest kilka sposobów. Jeden z nich polega na wywołaniu metody `new` z klasy `Proc`, innym jest wywołanie metody lambda albo proc z modułu `Kernel`. Tak przy okazji, wywołanie metody lambda lub proc jest preferowane w stosunku do `Proc.new`, ponieważ `lambda` i `proc` przeprowadzają kontrolę parametrów.

Listing 2.4 ilustruje, jak utworzyć obiekt `Proc` na oba sposoby.

Listing 2.4. proc.rb

```
#!/usr/bin/env/ruby

count = Proc.new { [1,2,3,4,5].each do |i| print i end; puts }
your_proc = lambda { puts "Lurch: 'Pani wzywała?'" }
my_proc = proc { puts "Morticia: 'Kto to był, Lurch?'" }

# Jakiego typu obiekty utworzyliśmy?
puts count.class, your_proc.class, my_proc.class

# Wywołanie wszystkich procedur
count.call
your_proc.call
my_proc.call
```

Po poinformowaniu, że wszystkie utworzone obiekty są obiektami `Proc`, program wyświetla poniższy tekst, wywołując po kolei każdą procedurę za pomocą metody `call`:

```
12345
Lurch: 'Pani wzywała?'
Morticia: 'Kto to był, Lurch?'
```

Metoda może zostać wywołana z blokiem; zwróci wtedy wynik wykonania bloku, nawet jeśli sama metoda nie przyjmuje żadnych argumentów. Przypominam, że blok musi zawsze być skojarzony z wywołaniem metody.

Możemy też „przekonać" metodę do skonwertowania w locie skojarzonego z nią bloku na obiekt Proc. W tym celu musimy utworzyć argument metody poprzedzony znakiem `&`. Ilustruje to listing 2.5.

Listing 2.5. return_block_proc.rb

```
#!/usr/local/bin/ruby

def return_block
  yield
end

def return_proc( &proc )
  yield
end

return_block { puts "Mam blok!" }
return_proc { puts "Mam blok, konwertuję na Proc!" }
```

Wyjście będzie wyglądać tak:

```
Mam blok!
Mam blok, konwertuję na Proc!
```

Metoda `return_block` nie ma argumentów i zawiera jedynie instrukcję `yield`. Zadaniem `yield` jest ponownie wykonanie bloku, gdy ten zostanie przekazany do metody. Ogromnie zwiększa to uniwersalność starej dobrej metody.

Następna metoda, `return_proc`, przyjmuje jeden argument — `&proc`. Gdy nazwa argumentu metody jest poprzedzona `&`, metoda przyjmie blok (jeśli został przekazany) i skonwertuje na obiekt Proc. Metoda zawiera w treści instrukcję `yield`, więc wykonuje blok będący zarazem procedurą, bez potrzeby użycia metody `call` obiektu `Proc`.

Symbole

Język Ruby zawiera specjalny obiekt nazwany **symbolem**. Na razie wystarczy nam wiedzieć o symbolach tyle, że są jakby pojemnikami na identyfikatory i łańcuchy. Symbol można rozpoznać po tym, że zawsze jest poprzedzany dwukropkiem (`:`).

Symbolu nie tworzy się bezpośrednio przez przypisanie do niego wartości. Aby utworzyć symbol, należy wywołać metodę `_sym` lub `intern` z łańcuchem lub przypisać symbol do symbolu. W celu zilustrowania tego przekształćmy łańcuch na symbol i z powrotem na łańcuch.

```
name = "Matz"
name.to_sym #=> ":Matz"
:Matz.id2name #=> "Matz"
name == :Matz.id2name #=> true
```

Zaczyna być niepokojąco zagmatwane? Wiem, że symbole mogą wyglądać trochę niejasno. Są dość abstrakcyjne, ponieważ nie wiemy tak naprawdę, co dzieje się „pod maską" interpretera Ruby. Patrząc z wierzchu widzimy, że zawartość łańcucha `name` została magicznie przekształcona na etykietę symbolu. No i co?

No i to, że od chwili utworzenia symbolu tylko jedna kopia symbolu jest przechowywana pod pojedynczym adresem pamięci, dopóki program pozostaje uruchomiony. Z tego powodu Ruby nie tworzy jednej kopii za drugą, lecz zawsze odwołuje się do tego jednego adresu w pamięci. Zwiększa to wydajność programów w Ruby, ponieważ zajmują mniej pamięci.

Ruby on Rails używa mnóstwa symboli, a w miarę poznawania języka Czytelnik zapewne również zacznie korzystać z wielu z nich. W istocie Ruby wewnętrznie wykorzystuje ogromną liczbę symboli. Aby się o tym przekonać, wystarczy wykonać następującą instrukcję Ruby:

```
Symbol.all_symbols
```

Dostaniemy listę ponad 1000 symboli!

Programistom dobrze znającym język C# lub Java pomoże taka analogia: symbole w języku Ruby są jak łańcuchy interned, przechowywane w puli łańcuchów intern.

Obsługa wyjątków

Ruby, podobnie jak Java, C++ i inne języki programowania, udostępnia obsługę wyjątków. Wyjątek występuje, gdy program wykona coś nieprzewidzianego i zostanie przerwany jego poprawny przepływ. Ruby jest przygotowany do radzenia sobie z takimi problemami, lecz możemy zająć się nimi na własny sposób, wykorzystując obsługę wyjątków.

W językach Java i C++ stosowane są bloki `try`; w Ruby użyjemy po prostu bloku `begin`. Odpowiednikiem instrukcji `catch` z Javy i C++ są w Ruby instrukcje `rescue`. Tam, gdzie w Javie używany jest warunek `finally`, Ruby używa warunku `ensure`.

Korzystanie z obsługi wyjątków zostało opisane w rozdziale 10.

Dokumentacja języka Ruby

Gdy mówię o „dokumentacji języka Ruby", mam na myśli przede wszystkim dokumentację wygenerowaną przez RDoc (*http://rdoc.sourceforge.net*) — program wydostający dokumentację z plików źródłowych Ruby, napisanych zarówno w języku Ruby, jak i C.

Dokumentacja jest w plikach kodu źródłowego umieszczona w komentarzach i zakodowana tak, że RDoc może ją łatwo znaleźć. Na przykład, znaki równości (jak na przykład ===) na lewym marginesie oznaczają nagłówek, a tekst wcięty jest formatowany jako kod. RDoc może generować wyjście jako pliki HTML, XML, ri (*Ruby information*) lub pliki pomocy Windows (*.chm*).

Wygenerowana przez RDoc dokumentacja Ruby w formacie HTML jest dostępna pod adresem *http://www.ruby-doc.org/core*. W systemie, w którym została zainstalowana dokumentacja języka Ruby (a została, jeśli postąpiliśmy zgodnie z instrukcjami instalacji z rozdziału 1.), można wpisać w wierszu zachęty polecenie jak poniżej, aby otrzymać sformatowaną dokumentację:

```
ri Kernel.print
```

Wynik będzie wyglądać tak:

```
------------------------------------------------------------ Kernel#print
     print(obj, ...)    => nil
--------------------------------------------------------------------------
     Prints each object in turn to +$stdout+. If the output field
     separator (+$,+) is not +nil+, its contents will appear between
     each field. If the output record separator (+$\+) is not +nil+, it
     will be appended to the output. If no arguments are given, prints
     +$_+. Objects that aren't strings will be converted by calling
     their +to_s+ method.

        print "cat", [1,2,3], 99, "\n"
        $, = ", "
        $\ = "\n"
        print "cat", [1,2,3], 99

     _produces:_

        cat12399
        cat, 1, 2, 3, 99
```

W rozdziale 10. zostały zawarte instrukcje, jak tworzyć dokumentację za pomocą RDoc.

Pytania sprawdzające

1. Jaka jest jedna z podstawowych różnic pomiędzy klasą a modułem?
2. Jaki moduł zawiera klasa `Object`?
3. Jak wygląda składnia służąca do tworzenia bloków komentarzy?
4. Jaki znak specjalny rozpoczyna nazwę zmiennej instancji? Zmiennej klasy? Zmiennej globalnej?
5. Jaka podstawowa cecha wyróżnia stałą?
6. Co, zgodnie z konwencją, oznacza zakończenie metody znakiem zapytania?

7. Blok to coś w rodzaju nienazwanej _____.
8. Co to jest „proc”?
9. Co jest najważniejszą właściwością symbolu?
10. Co to jest RDoc?