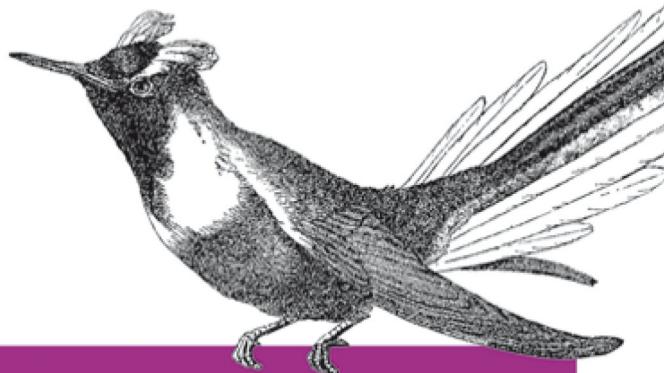


Poznaj możliwości języka Ruby!

Opisuje
Ruby 1.8 i 1.9



Ruby

Programowanie



HELION

O'REILLY®

David Flanagan, Yukihiko Matsumoto
Rysunki wykonane przez why the lucky stiffa

Tytuł oryginału: The Ruby Programming Language

Tłumaczenie: Łukasz Piwko

ISBN: 978-83-246-6182-4

© Helion S.A. 2009

Authorized translation of the English edition of The Ruby Programming Language © 2008
David Flanagan and Yukihiko Matsumoto. All rights reserved.

This translation is published and sold by permission of O'Reilly Media, Inc., the owner of all
rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form
or by any means, electronic or mechanical, including photocopying, recording or by any
information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości
lub fragmentu niniejszej publikacji w jakiekolwiek postaci jest zabronione.

Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki
na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich
niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi
bądź towarowymi ich właścicielami.

Autor oraz Wydawnictwo HELION dołożyli wszelkich starań, by zawarte
w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej
odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne
naruszenie praw patentowych lub autorskich. Autor oraz Wydawnictwo HELION
nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe
z wykorzystania informacji zawartych w książce.

Wydawnictwo HELION
ul. Kościuszki 1c, 44-100 GLIWICE
tel. 032 231 22 19, 032 230 98 63
e-mail: helion@helion.pl
WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Pliki z przykładami omawianymi w książce można znaleźć pod adresem:
<ftp://ftp.helion.pl/przyklady/rubypr.zip>

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

http://helion.pl/user/opinie?rubypr_ebook

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Poleć książkę na Facebook.com](#)
- [Kup w wersji papierowej](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Wstęp	7
1. Wprowadzenie	11
1.1. Krótki kurs języka Ruby	12
1.2. Wypróbuj język Ruby	20
1.3. Książka — informacje	24
1.4. Program rozwiązuje sudoku	25
2. Struktura i uruchamianie programów Ruby	31
2.1. Struktura leksykalna	32
2.2. Struktura syntaktyczna	39
2.3. Struktura plików	40
2.4. Kodowanie znaków	41
2.5. Wykonywanie programu	43
3. Typy danych i obiekty	45
3.1. Liczby	46
3.2. Tekst	50
3.3. Tablice	66
3.4. Tablice asocjacyjne	68
3.5. Zakresy	70
3.6. Symbole	72
3.7. Słowa kluczowe true, false i nil	73
3.8. Obiekty	73
4. Wyrażenia i operatory	85
4.1. Literały i literały słów kluczowych	86
4.2. Odwołania do zmiennych	87
4.3. Odwołania do stałych	88

4.4. Wywołania metod	89
4.5. Przypisania	91
4.6. Operatory	99
5. Instrukcje i przepływ sterowania	115
5.1. Instrukcje warunkowe	116
5.2. Pętle	124
5.3. Iteratory i obiekty przeliczalne	126
5.4. Bloki	136
5.5. Kontrola przepływu sterowania	141
5.6. Wyjątki i ich obsługa	148
5.7. Instrukcje BEGIN i END	158
5.8. Wątki, włókna i kontynuacje	159
6. Metody, obiekty klasy Proc, lambdy i domknięcia	165
6.1. Definiowanie prostych metod	167
6.2. Nazwy metod	170
6.3. Nawiasy w metodach	172
6.4. Argumenty metod	174
6.5. Obiekty proc i lambda	181
6.6. Domknięcia	188
6.7. Obiekty klasy Method	191
6.8. Programowanie funkcyjne	192
7. Klasy i moduły	201
7.1. Definiowanie prostej klasy	202
7.2. Widoczność metod — publiczne, chronione i prywatne	218
7.3. Tworzenie podklas i dziedziczenie	220
7.4. Tworzenie i inicjacja obiektów	227
7.5. Moduły	232
7.6. Funkcje load i require	236
7.7. Metody singletonowe i klasa eigenclass	240
7.8. Wyszukiwanie metod	242
7.9. Wyszukiwanie stałych	244
8. Refleksja i metaprogramowanie	247
8.1. Typy, klasy i moduły	248
8.2. Wykonywanie łańcuchów i bloków	250
8.3. Zmienne i stałe	252
8.4. Metody	254
8.5. Metody zwrotne	258
8.6. Śledzenie	259

8.7. Moduły ObjectSpace i GC	261
8.8. Niestandardowe struktury sterujące	262
8.9. Brakujące metody i stałe	264
8.10. Dynamiczne tworzenie metod	267
8.11. Tworzenie łańcuchów aliasów	269
8.12. Języki do wyspecjalizowanych zastosowań	274
9. Platforma Ruby	279
9.1. Łańcuchy	280
9.2. Wyrażenia regularne	285
9.3. Liczby i matematyka	295
9.4. Data i godzina	299
9.5. Kolekcje	300
9.6. Pliki i katalogi	320
9.7. Wejście i wyjście	325
9.8. Programowanie sieciowe	335
9.9. Wątki i współbieżność	340
10. Środowisko Ruby	355
10.1. Uruchamianie interpretera Ruby	356
10.2. Środowisko najwyższego poziomu	360
10.3. Praktyczne skróty do ekstrakcji i raportowania	368
10.4. Wywoływanie systemu operacyjnego	370
10.5. Bezpieczeństwo	374
Skorowidz	379

Wstęp

Niniejsza książka jest zaktualizowaną i rozszerzoną wersją książki *Ruby in a Nutshell* (O'Reilly) autorstwa Yukihiro Matsumoto, który jest szerzej znany pod pseudonimem „Matz”. Jest wzorowana na klasycznej pozycji *The C Programming Language* (Prentice Hall) Briana Kernighana i Dennisa Ritchiego. Jej autor stawia sobie za cel pełne udokumentowanie języka Ruby, ale w sposób bardziej przystępny niż ma to miejsce w specyfikacjach technicznych. Książka przeznaczona jest zarówno dla doświadczonych programistów, którzy zaczynają naukę języka Ruby, jak również dla znających ten język, ale pragnących lepiej go zrozumieć i nauczyć się nim posługiwać po mistrzowsku.

Opis struktury i organizacji niniejszej książki znajduje się w rozdziale 1.

Podziękowania

David Flanagan

Na samym początku pragnę podziękować „Matzowi” za to, że zaprojektował ten znakomity język, za pomoc w opanowaniu jego tajników, jak również za książkę *Ruby in a Nutshell*, na podstawie której powstała niniejsza publikacja.

Podziękowania należą się także następującym osobom:

- „why the lucky stiff” — za cudowne rysunki, które są ozdobą stron tej książki (znajdują się na stronie tytułowej każdego rozdziału) i oczywiście za jego własną książkę na temat języka Ruby pod tytułem *Why's (poignant) guide to Ruby*, którą można znaleźć pod adresem <http://www.poignantguide.net/ruby/>.
- Moim redaktorom merytorycznym: Davidowi A. Blackowi — dyrektorowi Ruby Power and Light, LLC (<http://www.rubypal.com>), Charlesowi Oliverowi Nutterowi z zespołu JRuby (<http://www.jruby.org>) w firmie Sun Microsystems, Shyouhei Urabe — maintainerowi gałęzi 1.8.6 języka Ruby i Kenowi Cooperowi. Ich komentarze wpłynęły na poprawę jakości i przejrzystości niniejszej książki. Oczywiście wszelkie błędy, które w niej pozostały, są moje.
- Mojemu redaktorowi Mike'owi Loukidesowi za ciągłe nakłanianie i zachęcanie mnie do napisania tej książki oraz za okazaną mi cierpliwość, kiedy ją pisałem.

Na koniec dziękuję mojej rodzinie.

— David Flanagan

<http://www.davidflanagan.com>

Styczeń 2008

Yukihiro Matsumoto

Poza osobami, które wymienił David (z wyjątkiem mnie), dziękuję za pomoc członkom społeczności z całego świata, a zwłaszcza z Japonii. Są to między innymi: Koichi Sasada, Nobuyoshi Nakada, Akira Tanaka, Shugo Maeda, Usaku Nakamura i Shyouhei Urabe (kolęność nie gra roli).

Na koniec dziękuję mojej rodzinie, która — mam nadzieję — wybaczy swojemu mężowi i ojcu, że poświęcił swój czas na pracę nad językiem Ruby.

— Yukihiro Matsumoto

Styczeń 2008

Konwencje typograficzne

W niniejszej książce zastosowano następujące konwencje typograficzne:

Pismo pochylone

Nowe terminy, adresy URL, adresy e-mail, nazwy plików i rozszerzenia plików.

Pismo o stałej szerokości znaków

Listingi kodu źródłowego programów, elementy programów znajdujące się wewnątrz zwykłego tekstu, jak nazwy zmiennych lub funkcji, typy danych, zmienne środowiskowe, instrukcje i słowa kluczowe.

Pogrubione pismo o stałej szerokości znaków

Polecenia lub innego rodzaju tekst, który powinien zostać dokładnie napisany przez użytkownika.

Pismo pochylone o stałej szerokości znaków

Tekst, który powinien zostać zastąpiony przez wartości użytkownika lub wartości wynikające z kontekstu.

Zasady korzystania z przykładów kodu

Niniejsza książka ma na celu pomóc czytelnikowi wykonać jego pracę. Ogólnie prezentowane w niej przykłady kodu można wykorzystywać we własnych programach i dokumentacji. Nie musisz pytać nas o zgodę, jeśli nie kopujesz dużej porcji kodu. Na przykład wykorzystanie kilku fragmentów kodu z niniejszej książki nie wymaga naszego pozwolenia. Sprzedaż lub dystrybucja płyt CD z przykładami z książek wydawnictw O'Reilly i Helion wymaga

zezwolenia. Odpowiedź na pytanie udzielona poprzez zacytowanie fragmentu tekstu i kodu z tej książki nie wymaga zezwolenia. Wykorzystanie dużej ilości kodu z niniejszej książki w dokumentacji własnego produktu wymaga uzyskania zezwolenia.

Przykładowe kody są dostępne do pobrania pod adresem

ftp://ftp.helion.pl/przyklady/rubypr.zip

Wprowadzenie



Ruby to dynamiczny język programowania ze złożoną, ale ekspresyjną gramatyką i główną biblioteką klas z bogatym i potężnym API. Jego twórcy inspirowali się językami Lisp, Smalltalk i Perl, ale jego gramatyka jest łatwa do zrozumienia także dla programistów C i Java™. Ruby to język czysto obiektowy, ale może również służyć do programowania proceduralnego i funkcjonalnego. Ponadto oferuje wiele narzędzi metaprogramowania, dzięki czemu może być używany do tworzenia języków do wyspecjalizowanych zastosowań (ang. *domain-specific language* — DSL).

„Matz” o języku Ruby

Yukihiro Matsumoto, znany szerzej w społeczności Ruby jako „Matz”, jest twórcą języka Ruby i autorem książki *Ruby in a Nutshell* (O'Reilly), której rozszerzoną wersją jest niniejsza pozycja. Matsumoto mówi:

Przed powstaniem języka Ruby znałem wiele języków, ale żaden z nich nie satysfakcjonował mnie w pełni. Były brzydsze, trudniejsze, bardziej skomplikowane lub prostsze, niż się spodziewałem. Postanowiłem stworzyć język programowania, który zaspokoiłby moje potrzeby programistyczne. Wiedziałem bardzo dużo na temat grupy docelowej tego nowego języka: to byłem ja. Ku mojnemu zaskoczeniu okazało się, że wielu programistów na całym świecie ma podobne odczucia do moich. Są zadowoleni, kiedy odkrywają język Ruby i piszą w nim programy.

Opracowując język Ruby, skoncentrowałem się na skróceniu czasu pisania szybszych programów. Wszystkie narzędzia tego języka, włącznie z narzędziami obiektowości, zostały zaprojektowane w taki sposób, aby działały tak, jak zwykły programista (np. ja) się tego spodziewa. Większość programistów uważa, że język ten jest elegancki i łatwy w użyciu, a pisanie w nim programów to sama przyjemność.

Często cytowaną myślą przewodnią, którą kierował się „Matz” podczas projektowania języka Ruby, jest poniższe zdanie:

Ruby został zaprojektowany, aby dawać szczęście programistom.

1.1. Krótki kurs języka Ruby

Niniejszy podrozdział zawiera krótki i nieco bezładny kurs po niektórych najbardziej interesujących funkcjach języka Ruby. Wszystkie poruszone tutaj zagadnienia są szczegółowo opisane w kolejnych rozdziałach. Ten podrozdział ma dać przedsmak tego, czym jest Ruby.

1.1.1. Język Ruby jest obiektowy

Zaczniemy od tego, że język Ruby jest **w pełni** obiektowy. Każda wartość jest obiektem, nawet proste literały liczbowe i wartości true, false oraz nil (nil to specjalne słowo kluczowe oznaczające brak wartości — jest to odpowiednik wartości null z innych języków programowania). W poniższym kodzie wywoływana jest metoda o nazwie class na rzecz tych wartości. Komentarze w języku Ruby oznaczane są znakiem #, a strzałki => w komentarzach oznaczają wartości zwarcane przez komentowane wiersze kodu (konwencja ta jest stosowana w całej książce):

```
1.class      # => Fixnum: liczba 1 jest typu Fixnum.  
0.0.class    # => Float: liczby zmienoprzecinkowe należą do klasy Float.  
true.class   # => TrueClass: true to instancja singleton klasy TrueClass.
```

```
false.class # => FalseClass.  
nil.class # => NilClass.
```

Wywołania metod i funkcji w wielu językach wymagają nawiasów. W powyższym kodzie nie ma ich jednak wcale. W języku Ruby nawiasy są zazwyczaj opcjonalne i często opuszczane, zwłaszcza w przypadku wywołań metod, które nie pobierają żadnych argumentów. Przez to, że nie ma nawiasów, wywołania metod wyglądają jak referencje do nazwanych pól lub zmiennych obiektów. Jest to zamierzona, ale faktem jest, iż język Ruby jest bardzo restrykcyjny, jeśli chodzi o hermetyzację obiektów. Nie ma dostępu do wnętrza obiektu z zewnątrz. Każda taka operacja musi odbywać się za pośrednictwem metody dostępowej, której przykładem jest zaprezentowana powyżej metoda `class`.

1.1.2. Bloki i iteratory

Możliwość wywoływania metod na rzecz liczb całkowitych nie jest żadną osobliwością w języku Ruby. Prawdę powiedziawszy, programiści Ruby robią to dość często.

```
3.times { print "Ruby! " }    # Drukuję napis "Ruby! Ruby! Ruby!".  
1.upto(9) {|x| print x }    # Drukuję "123456789".
```

`times` i `upto` to metody implementowane przez obiekty typu całkowitoliczbowego (`Integer`). Są to specjalne metody zwane **iteratorami**, które działają podobnie jak pętle. Kod znajdujący się między nawiasami klamrowymi nazywa się **blokiem**. Jest związany z wywoływaną metodą i służy jako ciało pętli. Iteratory i bloki są kolejnymi zasługującymi na uwagę własnościami języka Ruby. Mimo iż dostępna jest zwykła pętla `while`, częściej działania, które są powtarzane wiele razy, wykonywane są za pomocą metod.

Nie tylko liczby całkowite posiadają metody iteracyjne. Tablice (oraz podobne do nich obiekty wyliczeniowe) posiadają iterator o nazwie `each`; wywołuje on związkany z nim blok kodu jeden raz na rzecz każdego elementu tablicy. Przy każdym wywołaniu do bloku przekazywany jest jeden element z tablicy:

```
a = [3, 2, 1]      # To jest literal tablicowy.  
a[3] = a[2] - 1   # Aby pobrać i ustawić elementy tablicy, należy używać nawiasów kwadratowych.  
a.each do |elt|    # Metoda each jest iteratorem. Niniejszy blok posiada parametr elt.  
  print elt+1     # Drukuję "4321".  
end                # Niniejszy blok jest ograniczony słowami kluczowymi do i end zamiast nawiasów {}.
```

Na bazie `each` zdefiniowano wiele różnych przydatnych iteratorów:

```
a = [1,2,3,4]          # Tablica.  
b = a.map { |x| x*x }  # Podnoszenie elementów do kwadratu. b = [1,4,9,16].  
c = a.select { |x| x%2==0 } # Wybieranie liczb parzystych. c = [2,4].  
a.inject do |sum,x|    # Obliczanie sumy wszystkich elementów => 10.  
  sum + x  
end
```

Tablice asocjacyjne (ang. *hash*), podobnie jak tablice, są podstawową strukturą danych w języku Ruby. Jak sama nazwa wskazuje, opierają się na strukturze danych tablicy asocjacyjnej i służą do kojarzenia dowolnych obiektów będących kluczami z obiektami będącymi wartościami. Innymi słowy, tablica asocjacyjna wiąże obiekty wartości z obiektami kluczycy. Do sprawdzania i ustawiania wartości w tablicach asocjacyjnych, tak jak w zwykłych tablicach, służą nawiasy kwadratowe. W nawiasach tych zamiast indeksów w postaci liczb całkowitych powinny znajdować się obiekty. Klasa `Hash`, podobnie jak `Array`, definiuje także metodę iteracyjną `each`. Wywołuje ona związkany z nią blok kodu po jednym razie na rzecz każdej pary klucz-wartość w tablicy i — tym różni się od klasy `Array` — przekazuje klucz oraz wartość jako parametry do bloku:

```

h = {
  :jeden => 1,
  :dwa => 2
}
h[:jeden]
h[:trzy] = 3
h.each do |key,value|
  print "#{value}:#{key}; "
end
# Tablica asocacyjna wiążąca nazwy liczb z cyframi.
# Strzałki pokazują odwzorowania klucz => wartość.
# Dwukropiek oznacza literal typu Symbol.
# => 1. Dostęp do wartości poprzez klucz.
# Dodanie nowej pary klucz-wartość.
# Iteracja przez pary klucz-wartość.
# Zmienne zastępowane sąłańcuchami.
# Drukuję "1:jeden; 2:dwa; 3:trzy; ".

```

Kluczami w tablicach asocacyjnych w języku Ruby mogą być obiekty dowolnego typu, ale najczęściej używane są obiekty typu `Symbol`. Są one niezmienialnymi łańcuchami poddanymi działaniu metody `intern`. Można porównywać je pod względem tożsamości, a nie zawartości tekstowej (ponieważ dwa różne obiekty `Symbol` nigdy nie mają takiej samej treści).

Możliwość wiązania bloku kodu z wywołaniem metody jest podstawową i bardzo ważną funkcją języka Ruby. Mimo iż od razu nasuwającym się sposobem jej użycia jest tworzenie pętli, przydaje się ona także do jednokrotnego wykonywania bloku kodu. Na przykład:

```

File.open("data.txt") do |f| # Otwórz podany plik i przekaż strumień do bloku.
  line = f.readline           # Użyj strumienia do odczytu pliku.
end                         # Strumień jest automatycznie zamknięty w chwili napotkania słowa kluczowego end.
t = Thread.new do            # Wykonanie tego bloku w nowym wątku.
  File.read("data.txt")      # Odczyt pliku w tle.
end                          # Zawartość pliku dostępna jako wartość wątku.

```

Na marginesie warto zauważyc, że poprzednio przykład `Hash.each` zawierał niniejszy interesujący wiersz kodu:

```
print "#{value}:#{key}; "      # Zmienne zastępowane sąłańcuchami.
```

Łańcuchy znajdujące się w podwójnych cudzysłowach mogą zawierać dowolne wyrażenia Ruby otoczone znakami `#{ i }`. Wartość takiego wyrażenia jest konwertowana na łańcuch (za pomocą metody `to_s` obsługiwanej przez wszystkie obiekty), który zastępuje treść wyrażenia i otaczające je znaki. Taka zamiana wartości wyrażeń na łańcuchy zazwyczaj nazywa się **interpolacją łańcuchów**.

1.1.3. Wyrażenia i operatory

Składnia języka Ruby jest oparta na wyrażeniach. Struktury sterujące jak `if`, które w innych językach nazywają się instrukcjami, w Ruby są wyrażeniami. Mają wartości, podobnie jak inne prostsze wyrażenia, i możliwe jest napisanie następującego kodu:

```
minimum = if x < y then x else y end
```

Mimo że wszystkie instrukcje w języku Ruby są wyrażeniami, nie wszystkie zwracają użytkowe wartości. Na przykład pętla `while` i definicje metod to wyrażenia, które zwracają wartość `nil`.

Tak jak w większości języków programowania, wyrażenia w języku Ruby składają się z wartości i operatorów. Większość operatorów tego języka jest znana programistom zającym takie języki, jak C, Java, JavaScript lub inne podobne. Poniżej znajdują się przykłady niektórych operatorów Ruby:

<code>1 + 2</code>	<code>#=> 3: dodawanie.</code>
<code>1 * 2</code>	<code>#=> 2: mnożenie.</code>
<code>1 + 2 == 3</code>	<code>#=> true: ==porównywanie.</code>
<code>2 ** 1024</code>	<code># 2 do potęgi 1024: w języku Ruby liczby całkowite mogą być dowolnej wielkości.</code>

```

"Ruby" + " rzadzi!"      # => "Ruby rzadzi!": konkatenacjałańcuchów.
"Ruby! " * 3              # => "Ruby! Ruby! Ruby! ": powtarzaniełańcuchów.
"%d %s" % [3, "rubiny"]  # => "3 rubiny": formatowanie w stylu metody printf Pythona.
max = x > y ? x : y     # Operator warunkowy.

```

Wiele z operatorów w języku Ruby jest zaimplementowanych jako metody, a klasy mogą definiować (lub przeddefiniować) je wedle własnych potrzeb (nie ma natomiast możliwości definiowania całkiem nowych operatorów — rozpoznawany jest tylko ich ustalony zestaw). Jako przykład warto zauważyc, że operatory + i * inaczej działają na liczbach całkowitych iłańcuchach. We własnych klasach można zdefiniować te operatory w dowolny sposób. Innym dobrym przykładem jest operator <<. Klasy liczb całkowitych Fixnum i Bignum używają go do bitowego przesunięcia w lewo, podobnie jak w języku C. Natomiast inne klasy (tak jak w języku C++), takie jakłańcuchy, tablice i strumienie, używają tego operatora do operacji dołączania. Tworząc nową klasę, do której mogą być dołączane jakieś wartości, warto rozważyć zdefiniowanie operatora <<.

Jednym z operatorów, którego przesłonięcie daje największe korzyści, jest operator []. Klasy Array i Hash wykorzystują go do uzyskiwania dostępu do elementów przy użyciu indeksów i wartości asocjacyjnych za pomocą klucza. Możliwe jest nawet zdefiniowanie go jako metody pobierającej kilka argumentów, które są oddzielone przecinkami i umieszczone w nawiasach kwadratowych (klasa Array przyjmuje indeks i liczbę w nawiasach kwadratowych, które wyznaczają podtablicę określonej tablicy). Aby umożliwić używanie operatora nawiasów kwadratowych po lewej stronie wyrażenia przypisania, można zdefiniować odpowiedni operator []=. Wartość znajdująca się po prawej stronie przypisania zostanie przekazana jako ostatni argument do metody implementującej ten operator.

1.1.4. Metody

Metody definiuje się, używając słowa kluczowego def. Wartość zwrotna metody odpowiada wartości ostatniego wyrażenia w jej ciele:

```

def square(x)    # Definicja metody o nazwie square z jednym parametrem x.
  x*x            # Zwraca wartość parametru x podniesionego do kwadratu.
end              # Koniec metody.

```

Kiedy metoda jak ta powyżej jest zdefiniowana na zewnątrz klasy lub modułu, jest ona w rzeczywistości funkcją globalną, a nie metodą wywoływaną na rzecz obiektu (natomiast z technicznego punktu widzenia staje się ona prywatną metodą klasy Object). Metody mogą być definiowane także dla poszczególnych obiektów. Ich nazwy muszą być poprzedzone obiektami, dla których są definiowane. Nazywane są one **metodami singleton**. W taki właśnie sposób w języku Ruby definiuje się metody klasowe:

```

def Math.square(x)  # Definicja metody klasowej modulu Math.
  x*x
end

```

Moduł Math wchodzi w skład rdzennej biblioteki Ruby, a powyższy kod wstawia do niego nową metodę. Jest to kluczowa cecha tego języka — klasy i moduły są otwarte i można je modyfikować oraz rozszerzać w czasie pracy.

Parametry metod mogą mieć wartości domyślne, a metody mogą przyjmować dowolną liczbę argumentów.

1.1.5. Przypisywanie

Operator = (którego nie można przesłonić) przypisuje wartość do zmiennej:

```
x = 1
```

Przypisanie można połączyć z innymi operatorami, jak + czy -:

```
x += 1      # Zwiększenie x: warto zauważać, że w języku Ruby nie ma operatora ++.  
y -= 1      # Zmniejszenie y: operatora -- również nie ma.
```

W języku Ruby możliwe jest przypisanie równolegle więcej niż jednej wartości do więcej niż jednej zmiennej w jednym wyrażeniu przypisania:

```
x, y = 1, 2      # To samo co x = 1; y = 2.  
a, b = b, a      # Zamiana wartości dwóch zmiennych.  
x, y, z = [1, 2, 3] # Elementy tablic automatycznie przypisywane do zmiennych.
```

Metody w języku Ruby mogą zwracać więcej niż jedną wartość. W połączeniu z nimi przydatne jest przypisywanie równolegle. Na przykład:

```
# Definicja metody konwertującej współrzędne układu kartezjańskiego (x, y) na współrzędne biegunkowe.  
def polar(x,y)  
    theta = Math.atan2(y,x)      # Obliczanie kąta.  
    r = Math.hypot(x,y)         # Obliczanie odległości.  
    [r, theta]                 # Ostatnie wyrażenie określa wartość zwrotną.  
end  
# Użycie powyższej metody z przypisaniem równoległy.  
distance, angle = polar(2,2)
```

Metody kończące się znakiem równości (=) są wyjątkowe, ponieważ mogą być wywoływane za pomocą składni przypisania. Jeśli obiekt o posiada metodę o nazwie `x=`, to poniższe dwa wiersze kodu robią dokładnie to samo:

```
o.x=(1)          # Normalna składnia wywołania metody.  
o.x = 1          # Wywołanie metody przez przypisanie.
```

1.1.6. Znaki interpunkcyjne jako przyrostki i przedrostki

Wiadomo już, że metody, których nazwy kończą się znakiem =, można wywoływać przez wyrażenie przypisania. Nazwy metod w języku Ruby mogą kończyć się także znakiem zapytania lub wykrzyknikiem. Znak zapytania oznacza predykaty — metody zwracające wartości logiczne (typu Boolean). Na przykład w klasach Array i Hash znajdują się metody o nazwie `empty?`; sprawdzają one, czy określona struktura danych posiada jakieśkolwiek elementy. Wykrzyknik na końcu nazwy metody oznacza, że podczas jej stosowania należy zachować szczególną ostrożność. Kilka klas w języku Ruby posiada pary definicji metod o takich samych nazwach, z tym, że jedna z nich kończy się wykrzyknikiem, a druga nie. Najczęściej metoda bez wykrzyknika zwraca zmodyfikowaną kopię obiektu, na rzecz którego została wywołana, a wersja z wykrzyknikiem jest mutatorem modyfikującym sam obiekt. Na przykład w klasie Array znajdują się metody `sort` i `sort!`.

Poza znakami na końcu nazw można też spotkać znaki interpunkcyjne na początku nazw zmiennych. Zmienne globalne mają przedrostek \$, zmienne obiektowe — @, a zmienne klasowe — @@. Przyzwyczajenie się do tych przedrostków może zająć nieco czasu, ale w końcu każdy odkryje, że możliwość określenia zasięgu zmiennej na podstawie jej nazwy jest bardzo pożyteczną cechą. Przedrostki są potrzebne, aby zapobiec wieloznaczności bardzo elastycznej gramatyki języka Ruby. Można je traktować jako cenę, którą trzeba zapłacić za możliwość pomijania nawiasów wokół wywołań metod.

1.1.7. Wyrażenia regularne i zakresy

Wiadomo już, że podstawowymi strukturami danych w języku Ruby są tablice jednowymiarowe i tablice asocjacyjne. Zaprezentowane zostały też przykłady użycia liczb i łańcuchów. Są jeszcze dwa typy danych, które zasługują na uwagę. Obiekt typu Regexp (wyrażenie regularne) definiuje wzorzec tekstowy i posiada metody pozwalające określić, czy dany łańcuch pasuje do tego wzorca, czy nie. Obiekt typu Range (zakres) reprezentuje wartości (zazwyczaj liczby całkowite) zawierające się między dwoma punktami końcowymi. Wyrażenia regularne i zakresy w języku Ruby mają składnię literałową:

```
/[Rr]uby/      # Pasuje do łańcucha "Ruby" i "ruby".  
/\d{5}/        # Pasuje do pięciu kolejnych cyfr.  
1..3          # Wszystkie x, gdzie 1 <= x <= 3.  
1...3         # Wszystkie x, gdzie 1 <= x < 3.
```

Obiekty typu Regexp i Range definiują zwykły operator == służący do porównywania oraz dodatkowo operator === przeznaczony do sprawdzania dopasowań i członkostwa. Instrukcja case w języku Ruby (podobnie jak switch w C lub Javie) porównuje swoje wyrażenie z każdym z możliwych przypadków za pomocą operatora ===, dlatego jest on często nazywany **operatorem równości przypadków** (ang. *case equality operator*). Prowadzi to do następujących instrukcji warunkowych:

```
# Określa nazwę pokolenia w USA na podstawie daty urodzenia.  
# Wyrażenie case testuje zakresy za pomocą operatora ===.  
generation = case birthyear  
  when 1946..1963: "Baby Boomer"  
  when 1964..1976: "Pokolenie X"  
  when 1978..2000: "Pokolenie Y"  
  else nil  
end  
  
# Metoda prosząca użytkownika o potwierdzenie czegoś.  
def are_you_sure?  
  while true  
    print "Na pewno? [t/n]: "  
    response = gets  
    case response  
    when /^[tT]/  
      return true  
    when /^[nN]/, /^$/  
      return false  
    end  
  end  
end
```

1.1.8. Klasy i moduły

Klasa to zestaw wzajemnie powiązanych metod mogących modyfikować stan obiektu. Jest on określany przez zmienne obiektowe — takie, których nazwy zaczynają się od znaku @ i są właściwe konkretnemu obiektowi. Poniżej znajduje się przykładowa definicja klasy o nazwie Sequence. Demonstруje ona sposób pisania metod iteracyjnych i definiowania operatorów.

```
#  
# Niniejsza klasa reprezentuje szereg liczb charakteryzowanych przez trzy  
# parametry: from, to i by. Liczby x w szeregu podlegają następującym  
# dwóm ograniczeniom:  
#  
#   from <= x <= to  
#   x = from + n*by, gdzie n jest liczbą całkowitą  
#
```

```

class Sequence
  # Jest to klasa wyliczeniowa. Poniżej znajduje się definicja iteratora each.
  include Enumerable # Dodała metody wymienionego modułu do tej klasy.
  # Metoda initialize jest wyjątkowa. Jest automatycznie wywoływana w celu
  # inicjalizacji nowo utworzonych egzemplarzy klasy.
  def initialize(from, to, by)
    # Zapisanie parametrów w zmiennych obiektowych do późniejszego użycia.
    @from, @to, @by = from, to, by # Przypisanie równolegle i przedrostek @.
  end
  # Iterator wymagany przez moduł Enumerable.
  def each
    x = @from      # Rozpoczęcie w punkcie startowym.
    while x <= @to # Jeśli nie osiągnięto jeszcze końca,
      yield x      # przekazanie x do bloku związanego z iteratorem.
      x += @by     # Zwiększenie x.
    end
  end
  # Definicja metody length (tak jak tablice) zwracającej liczbę
  # wartości w szeregu.
  def length
    return 0 if @from > @to          # if użyte jako modyfikator instrukcji.
    Integer((@to - @from) / @by) + 1 # Obliczenie i zwrócenie długości szeregu.
  end
  # Definicja innej nazwy dla tej samej metody.
  # W języku Ruby metody często mają po kilka nazw.
  alias size length # size jest od tej pory synonimem nazwy length.
  # Przesłonięcie operatora dostępu do tablicy w celu umożliwienia dostępu swobodnego do szeregu.
  def[](index)
    return nil if index < 0 # Zwrot wartości nil dla indeksów ujemnych.
    v = @from + index * @by # Obliczenie wartości.
    if v <= @to            # Jeśli należy do szeregu,
      v                   # jest zwracana.
    else                  # W przeciwnym przypadku
      nil                 # zwrot wartości nil.
    end
  end
  # Przesłonięcie operatorów arytmetycznych, aby zwracały nowe obiekty typu Sequence.
  def *(factor)
    Sequence.new(@from * factor, @to * factor, @by * factor)
  end
  def +(offset)
    Sequence.new(@from + offset, @to + offset, @by)
  end
end

```

Poniżej znajduje się fragment programu, który wykorzystuje klasę Sequence:

```

s = Sequence.new(1, 10, 2) # Od 1 do 10 co 2.
s.each { |x| print x }      # Drukuję "13579".
print s[s.size - 1]         # Drukuję 9.
t = (s + 1) * 2             # Od 4 do 22 co 4.

```

Kluczowym elementem klasy Sequence jest jej iterator each. Jeśli nie trzeba definiować metod iteracyjnej, nie ma potrzeby definiowania całej klasy. W zamian można napisać metodę iteracyjną, która przyjmuje parametry from, to i by. Zamiast czynić ją globalną, lepiej zdefiniować ją w jej własnym module:

```

module Sequences # Nowy moduł.
  def self.fromtoby(from, to, by) # Metoda singleton modułu.
    x = from
    while x <= to
      yield x
      x += by
    end
  end
end

```

Mając tak zdefiniowany iterator, można napisać kod jak ten poniżej:

```
Sequences.fromtoby(1, 10, 2) { |x| print x } # Drukuje "13579".
```

Taki iterator powoduje, że nie ma konieczności tworzenia obiektu typu Sequence, aby iterować po szeregu liczb. Jednak nazwa tej metody jest dosyć długa, a składnia jej wywołania nie jest satysfakcyjną. To, co jest naprawdę potrzebne, to sposób iteracji obiektów typu Range w krokach innych niż co jeden. Jedną z zadziwiających własności języka Ruby jest to, że jego klasy, nawet te wbudowane rdzenne, są **otwarte** — każdy program może do nich dodawać metody. Dlatego możliwe jest definiowanie nowych metod iteracyjnych dla zakresów:

```
class Range
  def by(step)
    x = self.begin
    if exclude_end?
      while x < self.end
        yield x
        x += step
      end
    else
      while x <= self.end
        yield x
        x += step
      end
    end
  end
  # Przykłady:
  (0..10).by(2) { |x| print x } # Drukuje "0246810".
  (0...10).by(2) { |x| print x } # Drukuje "02468".
```

Otwarcie istniejącej klasy w celu jej rozszerzenia.
Definicja iteratora o nazwie by.
Rozpoczęcie w jednym punkcie końcowym zakresu.
Dla... zakresów, które nie wliczają końca.
Test przy użyciu operatora <.
W przeciwnym przypadku dla... zakresów, które zawierają end.
Test przy użyciu operatora <=.

Koniec definicji metody.
Koniec modyfikacji klasy.

Niniejsza metoda by jest wygodna w użyciu, ale niepotrzebna. W klasie Range znajduje się już metoda o nazwie step, która jest używana w tym samym celu. Rzadkie API Ruby jest bardzo bogate. Warto poświęcić nieco czasu na jego przestudiowanie (zobacz rozdział 9.), aby nie pisać metod, które są już zaimplementowane!

1.1.9. Niespodzianki w języku Ruby

Każdy język programowania posiada pułapki, w które wpadają początkujący programiści. Poniżej opisane są dwie zaskakujące własności języka Ruby.

Łańcuchy w języku Ruby są modyfikowalne. To może być zaskoczeniem zwłaszcza dla programistów Javy. Operator []= pozwala na zmianę znaków w łańcuchu, a także na wstawianie, usuwanie i podmienianie podłańcuchów. Operator << pozwala na dołączanie elementów do łańcucha, a klasa String zawiera definicje wielu innych metod modyfikujących łańcuchy na miejscu. Ponieważ łańcuchy są modyfikowalne, literaly łańcuchowe w programie nie są unikatowymi obiektami. Jeśli literał łańcuchowy znajdzie się w pętli, przy każdej iteracji da on w wyniku nowy obiekt. Aby zapobiec wszelkim modyfikacjom łańcucha (lub dowolnego obiektu) w przyszłości, należy na jego rzecz wywołać metodę freeze.

Instrukcje warunkowe i pętle w języku Ruby (np. if i while) na podstawie wyrażeń warunkowych podejmują decyzję, które odgałęzienie wykonać lub czy kontynuować działanie. Wyrażenia warunkowe często mają wartość true lub false, ale nie jest to wymagane. Wartość nil jest traktowana jako false, a **wszystkie pozostałe jako true**. To z pewnością zaskoczy programistów języka C, w którym 0 jest odpowiednikiem wartości false, i JavaScript, gdzie odpowiednikiem false jest pusty łańcuch.

1.2. Wypróbij język Ruby

Mamy nadzieję, że ten krótki przewodnik po kluczowych cechach języka Ruby wzbudził zainteresowanie w Czytelniku i że ma on chęć wypróbować ten język. W tym celu potrzebny jest interpreter języka Ruby oraz umiejętność posługiwania się trzema narzędziami: *irb*, *ri* oraz *gem*, które są dołączone do interpretera. Niniejszy podrozdział zawiera informacje, skąd je wziąć i jak ich używać.

1.2.1. Interpreter Ruby

Oficjalna witryna internetowa języka Ruby znajduje się pod adresem <http://www.ruby-lang.org>. Jeśli ktoś nie ma jeszcze zainstalowanego interpretera Ruby, może kliknąć odnośnik *Pobierz* w witrynie *ruby-lang.org*. Znajdują się tam instrukcje dotyczące pobierania i instalacji standardowej implementacji referencyjnej interpretera Ruby opartej na języku C.

Po zainstalowaniu interpretera można go wywołać za pomocą polecenia `ruby`:

```
% ruby -e 'puts "witaj!"'  
witaj!
```

Opcja wiersza poleceń `-e` powoduje wykonanie przez interpreter jednego wiersza kodu, który należy podać jako argument. Najczęściej kod Ruby umieszcza się jednak w pliku i nakazuje interpreterowi jego wykonanie:

```
% ruby hello.rb  
hello world!
```

Inne implementacje języka Ruby

Ze względu na brak formalnej specyfikacji języka Ruby interpreter dostępny w witrynie <http://www.ruby-lang.org> jest implementacją referencyjną definiującą ten język. Czasami nazywa się ją MRI (czyli *Matz's Ruby Implementation*). W języku Ruby 1.9 oryginalny interpreter MRI został połączony z YARV (*Yet Another Ruby Virtual Machine*). W wyniku tego powstała nowa implementacja referencyjna wykonująca wewnętrzną komplikację do kodu bajtowego, a następnie uruchamiająca go w maszynie wirtualnej.

Jednak implementacja referencyjna nie jest jedyną dostępną. W chwili pisania niniejszych słów istniała alternatywna implementacja (JRuby) i kilka innych w fazie rozwoju.

JRuby

JRuby to implementacja Ruby oparta na Javie. Można ją pobrać pod adresem <http://jruby.org>. W chwili pisania tej książki jej aktualna wersja miała numer 1.1 i była zgodna z Ruby 1.8. Zanim książka ta pojawi się w księgarniach, może być już dostępna wersja JRuby zgodna z Ruby 1.9. JRuby to projekt open source opracowywany głównie w firmie Sun Microsystems.

IronRuby

IronRuby to implementacja Ruby firmy Microsoft przeznaczona dla platformy .NET i DLR (*Dynamic Language Runtime*). Kod źródłowy IronRuby jest dostępny na licencji Microsoft Permissive License. W chwili pisania tego tekstu IronRuby nie osiągnął jeszcze wersji 1.0. Witryna projektu znajduje się pod adresem <http://www.ironruby.net>.

Rubinius

Rubinius to projekt open source, który jest opisywany jako „alternatywna implementacja Ruby napisana w dużej części w języku Ruby. Maszyna wirtualna Rubinius o nazwie shotgun jest luźno oparta na architekturze Smalltalk-80 VM”. W chwili pisania tego tekstu nie było jeszcze wersji 1.0 Rubiniusa.

Cardinal

Cardinal to implementacja Ruby przeznaczona do działania na maszynie wirtualnej Parrot (ma ona wspierać język Perl 6 i kilka innych dynamicznych języków). W chwili pisania tego tekstu nie było wersji 1.0 oraz projektów Parrot i Cardinal. Cardinal nie posiada własnej witryny. Jest dostępny jako część otwartego projektu Parrot pod adresem <http://www.parrotcode.org>.

1.2.2. Wyświetlanie danych wyjściowych

Aby wypróbować działanie języka Ruby, trzeba wiedzieć, jak wyświetlić dane wyjściowe; dzięki temu testowane programy będą mogły wydrukować swoje wyniki. Jednym ze sposobów jest zastosowanie funkcji `puts` — została ona użyta w prezentowanym wcześniej kodzie `witaj!`. Ujmując rzecz krótko, funkcja `puts` drukuje łańcuch tekstu w konsoli i dołącza do niego znak nowego wiersza (chyba że na końcu łańcucha już się taki znajduje). Jeśli do metody `puts` zostanie przekazany obiekt niebędący łańcuchem, wywołuje ona na jego rzecz metodę `to_s` i drukuje łańcuch przez nią zwrócony. Funkcja `print` robi mniej więcej to samo, ale nie dołącza znaku nowego wiersza. Wpisz na przykład poniższy program składający się z dwóch wierszy kodu w edytorze tekstowym i zapisz go w pliku o nazwie `count.rb`:

```
9.downto(1) { |n| print n }      # Brak znaku nowego wiersza pomiędzy liczbami.  
puts " start!"                  # Zakończenie znakiem nowego wiersza.
```

Aby uruchomić ten program za pomocą interpretera Ruby, wpisz poniższe polecenie:

```
% ruby count.rb
```

Powinno to dać następujący wynik:

```
987654321 start!
```

Przydatną alternatywą dla funkcji `puts` jest funkcja `p`. Ma ona nie tylko krótszą nazwę, lecz również konwertuje obiekty na łańcuchy za pomocą metody `inspect`, która czasami zwraca bardziej przyjazne dla programisty reprezentacje obiektów niż metoda `to_s`. Na przykład w przypadku drukowania tablicy funkcja `p` używa notacji literała tablicowego, natomiast `puts` drukuje po prostu każdy element tablicy w osobnym wierszu.

1.2.3. Interaktywność języka Ruby dzięki irb

`irb` (skrót od *interactive Ruby*) to powłoka Ruby. Każde wyrażenie wpisane w wierszu polecień Ruby zostanie obliczone, a jego wartość wyświetlona. Jest to często najprostszy sposób wypróbowywania własności języka, które są opisywane w niniejszej książce. Poniżej znajduje się przykładowa sesja `irb` z komentarzami:

```
$ irb --simple-prompt          # Uruchomienie irb w terminalu.  
>> 2**3                         # Próba potęgowania.  
=> 8                            # Wynik.  
>> "Ruby! " * 3                 # Próba powtarzania łańcuchów.  
=> "Ruby! Ruby! Ruby! "          # Wynik.  
>> 1.upto(3){|x| puts x }        # Próba iteratora.
```

```
1          # Trzy wiersze danych wyjściowych,  
2          # ponieważ funkcja puts została wywołana trzy razy.  
3  
=> 1      # Wartość zwrotna wywołania I.upto(3).  
>> quit    # Koniec działania irb.  
$          # Powrót do wiersza poleceń.
```

Ta przykładowa sesja demonstruje wszystko, co trzeba wiedzieć na temat irb, aby produktywnie wykorzystać go podczas eksploracji języka Ruby. Narzędzie to posiada kilka innych ważnych funkcji, wliczając podpowłoki (należy wpisać irb w wierszu poleceń, aby uruchomić podpowłokę) oraz opcje konfiguracji.

1.2.4. Przeglądanie dokumentacji Ruby za pomocą ri

Kolejne istotne narzędzie Ruby to przeglądarka dokumentacji ri¹. Po wpisaniu w wierszu poleceń ri i nazwy klasy, modułu lub metody języka Ruby wyświetli się odpowiednia dokumentacja. Można podać nazwę metody bez klasy, do której ona należy. W takim przypadku zostanie wyświetlona tylko lista wszystkich metod o podanej nazwie (chyba że istnieje tylko jedna taka metoda). Normalnie nazwę klasy lub modułu można oddzielić od nazwy metody kropką. Jeśli klasa zawiera definicje metody klasowej i metody obiektowej o takiej samej nazwie, do tej pierwszej należy odwoływać się przy użyciu znaków ::, a do drugiej za pomocą znaku #. Poniżej znajduje się kilka przykładowych wywołań narzędzia ri:

```
ri Array  
ri Array.sort  
ri Hash#each  
ri Math::sqrt
```

Dokumentacja wyświetlana przez narzędzie ri pochodzi ze specjalnie sformatowanych komentarzy znajdujących się w kodzie źródłowym Ruby. Szczegóły na ten temat można znaleźć w podrozdziale 2.1.1.2.

1.2.5. Zarządzanie pakietami przy użyciu narzędzia gem

System zarządzania pakietami w Ruby nosi nazwę RubyGems. Pakiety i moduły przesyłane za jego pomocą to gemy (klejnoty). Narzędzie to pozwala w łatwy sposób instalować oprogramowanie Ruby i automatycznie zarządzać skomplikowanymi powiązaniami pomiędzy pakietami.

Skrrypt umożliwiający komunikację z narzędziem RubyGems nosi nazwę gem. W Ruby 1.9 jest on dostępny, podobnie jak irb i ri. W Ruby 1.8 instaluje się go osobno — informacje na ten temat można znaleźć na stronie <http://rubygems.org>. Po zainstalowaniu programu gem można go używać następująco:

```
# gem install rails  
Successfully installed activesupport-1.4.4  
Successfully installed activerecord-1.15.5  
Successfully installed actionpack-1.13.5  
Successfully installed actionmailer-1.3.5  
Successfully installed actionwebservice-1.2.5  
Successfully installed rails-1.2.5
```

¹ Opinie co do znaczenia skrótu ri są podzielone. Zgłasiane rozwinięcia to: Ruby Index, Ruby Information i Ruby Interactive.

```
6 gems installed
Installing ri documentation for activesupport-1.4.4...
Installing ri documentation for activerecord-1.15.5...
...itd...
```

Jak widać, polecenie `gem install` powoduje zainstalowanie najnowszej wersji żądanego oprogramowania oraz dodatkowych gemów przez nie wymaganych. Skrypt gem obsługuje kilka innych przydatnych poleceń, np.:

```
gem list          # Wyświetla listę zainstalowanych gemów.
gem environment # Wyświetla informację o konfiguracji RubyGems.
gem update rails # Aktualizuje podany gem.
gem update        # Aktualizuje wszystkie zainstalowane gemy.
gem update --system # Aktualizuje narzędzie RubyGems.
gem uninstall rails # Usuwa zainstalowany gem.
```

W Ruby 1.8 zainstalowanych gemów nie można automatycznie załadować za pomocą metody Ruby `require` (więcej informacji na temat ładowania modułów kodu Ruby przy użyciu tej metody znajduje się w rozdziale 7.6). Pisząc program, który będzie używał modułów zainstalowanych jako gemy, należy wpierw załadować moduł `rubygems`. Niektóre dystrybucje Ruby 1.8 mają wstępnie skonfigurowaną bibliotekę RubyGems, ale może być konieczne pobranie i zainstalowanie jej we własnym zakresie. Dzięki załadowaniu modułu `rubygems` metoda `require` zmienia sposób działania. Zanim przeszuka bibliotekę standardową, najpierw przeszukuje wszystkie zainstalowane gemy. Można także automatycznie włączyć obsługę RubyGems, używając opcji wiersza poleceń `-rubygems`. Aby biblioteka RubyGems była ładowana przy każdym wywołaniu Ruby, należy dodać `-rubygems` do zmiennej środowiskowej `RUBYOPT`.

W Ruby 1.9 moduł `rubygems` wchodzi w skład biblioteki standardowej, ale nie jest już konieczny do ładowania gemów. Ruby 1.9 potrafi samodzielnie znaleźć zainstalowane gemy, dzięki czemu w programach ich używających nie trzeba umieszczać kodu `require 'rubygems'`.

Metoda `require` (zarówno w Ruby 1.8, jak i 1.9) ładuje zawsze najnowszą wersję zażdanego gema. Aby załadować wybraną wersję gema, można użyć metody `gem` przed metodą `require`. Znajduje ona określoną wersję gema i aktywuje ją, dzięki czemu wywołana później metoda `require` może ją załadować.

```
require 'rubygems'           # Niepotrzebne w Ruby 1.9.
gem 'RedCloth', '> 2.0', '< 4.0' # Aktywacja gema RedCloth w wersji 2.x lub 3.x.
require 'RedCloth'            # Załadowanie powyższego gema.
```

Więcej informacji na temat metody `require` i gemów znajduje się w rozdziale 7.6.1. Pełny opis narzędzia RubyGems, programu `gem` i modułu `rubygems` wykracza poza zakres tej książki. Dokumentację polecenia `gem` można wyświetlić za pomocą polecenia `gem help`. Szczegóły na temat metody `gem` wyświetla polecenie `ri gem`. Pełne informacje można znaleźć w dokumentacji pod adresem <http://rubygems.org>.

1.2.6. Więcej kursów Ruby

Niniejszy rozdział zaczął się kursem wprowadzającym do języka Ruby. Zaprezentowane do tej pory przykłady kodu można wypróbować, używając narzędzia `irb`. Jeśli potrzebujesz większej liczby kursów przed przejściem do bardziej formalnego opisu języka, na stronie <http://www.ruby-lang.org> znajdują się odnośniki do dwóch dobrych kursów. Jeden z nich oparty

jest na irb i nosi nazwę *Ruby in Twenty Minutes*. Drugi ma tytuł *Try Ruby!*. Jego zaletą jest to, że działa w przeglądarce i nie wymaga instalacji interpretera Ruby oraz narzędzia irb na własnym komputerze².

1.2.7. Zasoby sieciowe

Na stronie głównej Ruby (<http://www.ruby-lang.org>) znajdują się odnośniki do różnych innych zasobów na temat tego języka, takich jak: dokumentacja online, biblioteki, listy mailingowe, blogi, kanały IRC, grupy użytkowników i konferencje. Aby je znaleźć, należy kliknąć odpowiednie odnośniki — *Dokumentacja*, *Biblioteki* i *Społeczność*.

1.3. Książka — informacje

Jak wskazuje tytuł, niniejsza książka opisuje język Ruby. Aspiruje ona do miana dostępnego i pełnego źródła wiedzy na ten temat. Opisywane wersje języka to 1.8 i 1.9. Granica pomiędzy językiem a platformą Ruby jest niewyraźna, dlatego poza opisem samego języka znajduje się tu szczegółowy przegląd rdzennego API Ruby. Książka ta nie jest jednak podręcznikiem do API, a zatem nie opisuje w pełni rdzennych klas. Ponadto nie jest to publikacja na temat frameworków (np. Rails) ani narzędzi Ruby (jak rake lub gem).

Niniejszy rozdział kończy się przykładowym skomplikowanym programem opatrzonym bardzo dużą liczbą komentarzy. Kolejne rozdziały opisują Ruby od początku do końca.

- W rozdziale 2. zaprezentowana jest struktura leksykalna i syntaktyczna języka Ruby, w tym podstawowe tematy, jak zestaw znaków, rozpoznawanie wielkich liter i słowa zarezerwowane.
- W rozdziale 3. przedstawione są rodzaje danych — liczby,łańcuchy,zakresy,tablice itd. — którymi mogą operować programy Ruby, oraz podstawowe własności wszystkich obiektów Ruby.
- Rozdział 4. mówi o wyrażeniach podstawowych w Ruby — literalach, odwołaniach do zmiennych, **wywołaniach** metod i przypisaniach — oraz operatorach służących do łączenia wyrażeń podstawowych w złożone.
- W rozdziale 5. opisane są instrukcje warunkowe, pętle (w tym bloki i metody iteracyjne), wyjątki oraz pozostałe wyrażenia, które w innych językach są nazywane instrukcjami lub instrukcjami sterującymi.
- Rozdział 6. zawiera formalną dokumentację sposobów definiowania metod i składni wywołań oraz opisuje możliwe do wywołania obiekty znane jako proc i lambda. W rozdziale tym znajduje się także opis domknięć i technik programowania funkcyjnego w języku Ruby.
- W rozdziale 7. zaprezentowane jest definiowanie klas i modułów. Klasy są podstawą programowania obiektowego. Poza nimi w rozdziale tym opisane są: dziedziczenie, widoczność metod, moduły dodawalne (mixiny) oraz algorytm rozwijania nazw metod.
- Rozdział 8. przedstawia API Ruby pozwalające programowi na introspekcję oraz techniki metaprogramowania, które przy wykorzystaniu API ułatwiają programowanie. Zawiera on także przykład języka do wyspecjalizowanych zastosowań (DSL).

² Jeśli w witrynie Ruby nie ma odnośnika do kursu *Try Ruby!*, powinien znajdować się on pod adresem <http://tryruby.hobix.com>.

- W rozdziale 9. zademonstrowane są najważniejsze klasy i metody platformy Ruby wraz z przykładami kodu. Nie jest to podręcznik, a szczegółowy przegląd rdzennych klas. W rozdziale tym poruszone są tematy przetwarzania tekstu, obliczeń matematycznych, kolekcji (jak tablice jednowymiarowe i asocjacyjne), wejścia i wyjścia, sieci oraz wątków. Po przeczytaniu go poznasz możliwości platformy Ruby i będziesz w stanie samodzielnie korzystać z narzędzi ri lub dokumentacji online w celu znalezienia szczegółowych informacji.
- Rozdział 10. opisuje najwyższy poziom środowiska programistycznego Ruby, w tym zmienne i funkcje globalne, argumenty wiersza poleceń rozpoznawane przez interpreter Ruby oraz mechanizmy bezpieczeństwa.

1.3.1. Jak czytać tę książkę

Programowanie w języku Ruby jest łatwe, chociaż sam język nie jest prosty. Ponieważ niniejsza książka w pełni go opisuje, nie jest prosta (mamy jednak nadzieję, że łatwo się ją czyta i rozumie). Jest przeznaczona dla doświadczonych programistów, którzy chcą opanować język Ruby i są gotowi przeczytać ją uważnie i ze zrozumieniem, aby osiągnąć ten cel.

Tak jak każda książka na temat programowania, publikacja ta zawiera odwołania do wcześniejszych i dalszych fragmentów. Języki programowania nie są systemami liniowymi i nie ma możliwości opisania ich w ten sposób. Jak można się przekonać po układzie niniejszego rozdziału, w książce tej zastosowano podejście od dołu do góry — najpierw zostają omówione najprostsze elementy gramatyki, które przechodzą do coraz bardziej złożonych struktur syntaktycznych — od tokenów, poprzez wartości i wyrażenia, po instrukcje sterujące, metody i klasy. Jest to klasyczny sposób dokumentacji języków programowania, który jednak nie pozwala uniknąć problemu odwołań do dalszych części.

Niniejsza książka powinna być czytana od początku do końca, chociaż niektóre zaawansowane zagadnienia najlepiej przejrzeć tylko pobiędzie lub pominać całkiem za pierwszym razem. Będzie można je znacznie łatwiej zrozumieć, wracając do nich po przeczytaniu dalszych rozdziałów. Z drugiej strony nie należy bać się każdego odwołania. Wiele z nich to po prostu informacje, że więcej szczegółów zostanie podanych dalej. Odwołanie nie zawsze oznacza, że wiadomości znajdujące się w dalszej części są niezbędne do zrozumienia aktualnie prezentowanego materiału.

1.4. Program rozwiążający sudoku

Na końcu niniejszego rozdziału prezentujemy skomplikowany program, którego celem jest zilustrowanie, jak wygląda kod w języku Ruby. Wybraliśmy średniej wielkości program rozwiążający sudoku³, ponieważ pozwala on zademonstrować wiele własności języka Ruby. Nie próbuj zrozumieć wszystkiego, ale przeczytaj cały kod. Dzięki wielu komentarzom czytanie nie powinno być zbyt trudne.

³ Sudoku to łamigłówka logiczna w formie siatki kwadratów 9x9. Polega ona na wypełnieniu wszystkich pustych kwadratów cyframi od 1 do 9 w taki sposób, aby żaden wiersz, kolumna ani kwadrat 3x3 nie zawierały dwóch takich samych cyfr. Sudoku jest popularne w Japonii już od dłuższego czasu. Na świecie zyskało dużą popularność stosunkowo niedawno, około 2004 roku. Jeśli nie znasz jeszcze sudoku, przeczytaj artykuł w Wikipedii i spróbuj rozwiązać łamigłówkę w internecie pod adresem <http://websudoku.com>.

Listing 1.1. Program w języku Ruby rozwiążający sudoku

```
#  
# Niniejszy moduł definiuje klasę Sudoku::Puzzle reprezentującą siatkę 9x9  
# oraz klasy wyjątków zgłaszanych w wyniku podania nieprawidłowych danych i  
# zbyt restykcyjnych ograniczeń lamigłówki. Moduł ten definiuje także metodę  
# Sudoku.solve rozwiązującą lamigłówkę. Metoda solve używa metody  
# Sudoku.scan, która również jest tutaj zdefiniowana.  
#  
# Aby użyć tego modułu do rozwiązywania sudoku, potrzebny jest kod jak poniżej:  
#  
# require 'Sudoku'  
# puts Sudoku.solve(Sudoku::Puzzle.new(ARGV.readlines))  
#  
module Sudoku  
#  
# Klasa Sudoku::Puzzle reprezentuje stan lamigłówki sudoku 9x9.  
#  
# Niektóre definicje i terminy używane w tej implementacji:  
#  
# - Każdy element lamigłówki nazywany jest „komórką”.  
# - Numery wierszy i kolumn należą do przedziału od 0 do 8. Współrzędne [0,0]  
# odnoszą się do komórki w lewym górnym rogu.  
# - Dziewięć komórek 3x3 nazywa się blokami. Ich numery również  
# należą do przedziału od 0 do 8 w kolejności od lewej do prawej i z góry do dołu.  
# Blok w lewym górnym rogu ma numer 0. Blok w prawym górnym rogu ma numer 2.  
# Blok znajdujący się na środku ma numer 4. Blok znajdujący się w prawym dolnym  
# rogu ma numer 8.  
#  
# Do tworzenia nowego sudoku służy metoda Sudoku::Puzzle.new. Należy określić  
# stan początkowy za pomocą łańcucha lub tablicy łańcuchów. Łąncuch ten (lub łańcuchy)  
# powinien zawierać znaki od 1 do 9 określające wartości i znak . dla pustych komórek.  
# Biale znaki są ignorowane.  
#  
# Zapis i odczyt poszczególnych komórek odbywa się za pośrednictwem operatorów  
# [] i [[], które wymagają indeksów dwuwymiarowych [wiersz, kolumna].  
# Metody te używają liczb (nie znaków) od 0 do 9 w komórkach.  
# 0 reprezentuje nieznaną wartość.  
#  
# Predykat has_duplicates? zwraca wartość true, jeśli lamigłówka jest nieprawidłowa  
# ze względu na dwukrotne wystąpienie jakiejś cyfry w którejkolwiek kolumnie albo  
# którymkolwiek wierszu lub bloku.  
#  
# Metoda each_unknown jest iteratorem przechodzącym przez komórki  
# i wywołującym związaną z nim blok dla każdej komórki, której wartość nie jest znana.  
#  
# Metoda possible zwraca tablicę liczb całkowitych z przedziału 1...9.  
# Tylko elementy tej tablicy są dozwolonymi wartościami w określonej komórce.  
# Jeśli tablica ta jest pusta, lamigłówka nie może zostać rozwiązana.  
# Jeśli tablica ta ma tylko jeden element, musi on być  
# wartością tej komórki.  
#  
class Puzzle  
# Niniejsze stałe służą do translacji między zewnętrzną reprezentacją  
# łańcuchów lamigłówki a reprezentacją wewnętrzną.  
ASCII = ".123456789"  
BIN = "\000\001\002\003\004\005\006\007\010\011"  
# Metoda inicjująca klasy. Jest automatycznie wywoływana na rzecz  
# nowych egzemplarzy klasy Puzzle tworzących za pomocą metody Puzzle.new.  
# Wejściowa lamigłówka jest przekazywana jako tablica wierszy lub pojedynczy łańcuch.  
# Można używać cyfr ASCII od 1 do 9. Znak . oznacza niewiadomą komórkę.  
# Biale znaki, wliczając znaki nowego wiersza, są usuwane.  
def initialize(lines)  
    if (lines.respond_to? :join) # Jeśli argument przypomina tablicę wierszy,  
        s = lines.join # zostaną one połączone w jeden łańcuch.  
    else  
        s = lines.dup # W przeciwnym razie należy założyć, że istnieje łańcuch,  
    end
```

```

end
# Biale znaki w danych muszą zostać usunięte.
# Znak ! w nazwie metody gsub! oznacza, że jest to metoda mutacyjna, która
# bezpośrednio modyfikuje łańcuch, zamiast robić jego kopię.
s.gsub!(/\s/, "") # /s/ to wyrażenie regularne, które pasuje do każdego białego znaku.
# Jeśli dane wejściowe mają nieprawidłowy rozmiar, zgłoszony jest wyjątek.
# Należy zauważać, że użyto słowa unless w formie modyfikatora zamiast if.
raise Invalid, "Nieprawidłowy rozmiar siatki." unless s.size == 81

# Szukanie nieprawidłowych znaków i zapis lokalizacji pierwszego z nich.
# Należy zauważać, że przypisanie i sprawdzenie wartości odbywa się jednocześnie.
if i = s.index(/[^123456789\.]/)
  # Dобавление nieprawidłowego znaku do komunikatu o błędzie.
  # Zauważ wyrażenie w literale łańcuchowym ${}.
  raise Invalid, "Niedozwolony znak #{s[i,1]} "
end
# Poniższe dwa wiersze kodu konwertują łańcuch znaków ASCII
# na tablicę liczb całkowitych za pomocą dwóch metod String.
# Powstała w wyniku tego tablica zostaje zapisana w zmiennej obiektowej @grid.
# Zero reprezentuje nieznaną wartość.
s.tr!(ASCII, BIN) # Translacja znaków ASCII na bajty.
@grid = s.unpack('c*') # Wypakowanie bajtów do tablicy liczb.
# Upewnienie się, że wiersze, kolumny i bloki nie zawierają duplikatów.
raise Invalid, "W łamigłówce występują duplikaty" if has_duplicates?
end
# Zwróci stan łamigłówki jako łańcuch dziewięciu wierszy z dziewięcioma
# znakami w każdym (plus znak nowego wiersza).
def to_s
  # Niniejsza metoda została zaimplementowana w jednym wierszu, który odwraca
  # działanie metody initialize(). Pisanie tak zagęszczonego kodu raczej
  # nie należy do dobrego stylu programowania, ale demonstruje siłę
  # i ekspresję języka.
  #
  # Działanie tego kodu jest następujące:
  # (0..8).collect wywołuje kod w klamrach 9 razy - po jednym dla
  # każdego wiersza - i zapisuje wartość zwrotną tego kodu w tablicy.
  # Kod w klamrach pobiera podtablicę siatki reprezentującą
  # pojedynczy wiersz i wstawia jego liczby do łańcucha.
  # Metoda join() łączy elementy tablicy w pojedynczy łańcuch
  # ze znakami nowego wiersza między poszczególnymi cyframi. Metoda
  # tr() dokonuje translacji binarnej reprezentacji łańcucha na cyfry ASCII.
  (0..8).collect{|r| @grid[r*9..9].pack('c9')}.join("\n").tr(BIN,ASCII)
end
# Zwróci duplikat obiektu Puzzle.
# Ta metoda przesyłania Object.dup i kopiuje tablicę @grid.
def dup
  copy = super # Utworzenie płytkiej kopii za pomocą metody Object.dup.
  @grid = @grid.dup # Utworzenie nowej kopii danych wewnętrznych.
  copy # Zwrócenie skopiowanego obiektu.
end
# Przesłoń operator dostępu do tablicy, aby umożliwić dostęp do
# poszczególnych komórek. Łamigłówki są dwuwymiarowe, a więc muszą być
# indeksowane współrzędnymi wiersza i kolumny.
def [](row, col)
  # Konwersja dwuwymiarowych współrzędnych (row, col) na indeksy tablicy
  # jednowymiarowej oraz pobranie i zwrócenie wartości w komórce o takim indeksie.
  @grid[row*9 + col]
end
# Niniejsza metoda pozwala na użycie operatora dostępu do tablicy po lewej
# stronie operacji przypisania. Ustawia wartość komórki o współrzędnych
# (row, col) na wartość newvalue.
def []=(row, col, newvalue)
  # Jeżeli nowa wartość nie mieści się w przedziale 0-9, zgłoszony jest wyjątek.
  unless (0..9).include? newvalue
    raise Invalid, "Nieprawidłowa wartość w komórce"
  end

```

```

end
# Ustawienie odpowiedniego elementu wewnętrznej tablicy na tę wartość.
@grid[row*9 + col] = newvalue
end
# Ta tablica odwzorowuje jednowymiarowy indeks siatki na numer bloku.
# Jest on używany w poniższej metodzie. Nazwa BoxOfIndex zaczyna się od wielkiej litery,
# a więc jest to stała. Ponadto tablica została zamrożona, przez co nie
# można jej modyfikować.
BoxOfIndex = [
    0,0,0,1,1,1,2,2,2,0,0,0,1,1,1,2,2,2,0,0,0,1,1,1,2,2,2,
    3,3,3,4,4,4,5,5,5,3,3,3,4,4,4,5,5,5,3,3,3,4,4,4,5,5,5,
    6,6,6,7,7,7,7,8,8,8,6,6,6,7,7,7,8,8,8,6,6,6,7,7,7,8,8,8
].freeze
# Niniejsza metoda jest iteratorem lamiówk sudoku.
# Dla każdej komórki z nieznaną wartością metoda ta przekazuje
# numery wiersza, kolumny i bloku do bloku kodu
# związanego z tym iteratorem.
def each_unknown
  0..upto 8 do |row|                      # Każdy wiersz.
  0..upto 8 do |col|                      # Każda kolumna.
    index = row*9+col                     # Indeks komórki dla (row, col).
    next if @grid[index] != 0              # Kontynuuj, jeśli znana jest wartość komórki.
    box = BoxOfIndex[index]                # Sprawdź blok tej komórki.
    yield row, col, box                   # Wywołaj odpowiedni blok kodu.
  end
end
# Zwraca wartość true, jeśli któryś wiersz, kolumna lub blok zawiera duplikaty.
# W przeciwnym przypadku zwraca wartość false. W sudoku w wierszu, kolumnie
# ani bloku nie może być duplikatów. Dlatego wartość true oznacza złą lamiówkę.
def has_duplicates?
  # Metoda uniq! zwraca wartość nil, jeśli wszystkie elementy tablicy są unikatowe.
  # Jeśli metoda uniq! zwraca jakąś wartość, lamiówka zawiera duplikaty.
  0..upto(8) { |row| return true if rowdigits(row).uniq! }
  0..upto(8) { |col| return true if coldigits(col).uniq! }
  0..upto(8) { |box| return true if boxdigits(box).uniq! }

  false  # Jeśli wszystkie testy zakończyły się powodzeniem, lamiówka nie zawiera duplikatów.
end
# Ta tablica zawiera zestaw wszystkich cyfr sudoku. Jest używana poniżej.
AllDigits = [1, 2, 3, 4, 5, 6, 7, 8, 9].freeze
# Zwraca tablice wszystkich wartości, które mogą znaleźć się w komórce
# o współrzędnych (row, col), nie duplikując wartości w wierszu, kolumnie i bloku.
# Zauważ, że operator + w tablicach oznacza konkatencję, natomiast operator -
# wykonuje operację różnicy zbiorów.
def possible(row, col, box)
  AllDigits - (rowdigits(row) + coldigits(col) + boxdigits(box))
end
private  # Wszystkie metody za tym wierszem są prywatne dla klasy.
# Zwraca tablice wszystkich znanych wartości w określonym wierszu.
def rowdigits(row)
  # Wydobywa podtablicę reprezentującą ten wiersz i usuwa wszystkie zera.
  # Odejmowanie tablic jest operacją różnicy zbiorów z usuwaniem duplikatów.
  @grid[row*9..9] - [0]
end
# Zwraca tablicę wszystkich znanych wartości w określonej kolumnie.
def coldigits(col)
  result = []                                # Na początku jest pusta tablica.
  col.step(80, 9) { |i|                      # Pętla zaczyna działanie od col w krokach co 9 do 80.
    v = @grid[i]                            # Pobranie wartości komórki pod tym indeksem.
    result << v if (v != 0)                 # Jeśli jest różna od 0, zostaje dodana do tablicy.
  }
  result                                     # Zwrócenie tablicy.
end
# Odwzorowanie numeru bloku na indeks lewego górnego rogu tego bloku.

```

```

BoxToIndex = [0, 3, 6, 27, 30, 33, 54, 57, 60].freeze
# Zwraca tablicę wszystkich znanych wartości w określonym bloku.
def boxdigits(b)
    # Konwersja numeru bloku na indeks lewego górnego rogu tego bloku.
    i = BoxToIndex[b]
    # Zwraca tablicę wartości z usunięciem zer.
    [
        @grid[i],    @grid[i+1],    @grid[i+2],
        @grid[i+9],  @grid[i+10],   @grid[i+11],
        @grid[i+18], @grid[i+19],   @grid[i+20]
    ] - [0]
end
end # Koniec klasy Puzzle.
# Wyjątek tej klasy oznacza nieprawidłowe dane na wejściu.
class Invalid < StandardError
end
# Wyjątek tej klasy oznacza, że ograniczenia są zbyt restrykcyjne, przez co nie ma
# żadnego rozwiązania.
class Impossible < StandardError
end
#
# Niniejsza metoda skanuje lamigłówkę w poszukiwaniu nieznanych komórek, które
# mogą przyjąć tylko jedną z wartości. Jeśli zostaną takie znalezione, wstawia do nich wartości.
# Ponieważ ustawienie wartości w jednej komórce ma wpływ na możliwe wartości w innych
# komórkach, metoda ta kontynuuje skanowanie, aż przeskanuje całą lamigłówkę i nie
# znajdzie żadnej komórki, której wartość można ustawić.
#
# Niniejsza metoda zwraca trzy wartości. Jeśli rozwiąże lamigłówkę,
# zwraca trzy razy nil. W przeciwnym przypadku dwie pierwsze wartości określają wiersz
# i kolumnę, których komórka nie ma wartości. Trzecia wartość to zbiór
# wartości, które mogą znaleźć się w tej kolumnie i tym wierszu. Jest to minimalny
# zbiór możliwych wartości — nie ma nieznanej komórki w lamigłówce, która ma mniej
# możliwych wartości. Ta złożona wartość zwrotna pozwala na przydatną heurystykę
# w metodzie solve() — metoda ta może zgadywać wartości w komórkach, gdzie
# jest największe prawdopodobieństwo trafienia.
#
# Niniejsza metoda generuje wyjątek Impossible, jeśli znajdzie komórkę, dla której
# nie istnieje możliwa wartość. Może się to zdarzyć, kiedy ograniczenia są zbyt surowe
# lub jeśli znajdująca się niżej metoda solve() źle zgadła.
#
# Metoda ta modyfikuje określony obiekt Puzzle w miejscu.
# Jeśli has_duplicates? ma wartość false na wejściu, będzie mieć wartość false także na wyjściu.
#
def Sudoku.scan(puzzle)
    unchanged = false # Zmienna pętlowa.
    # Powtarzanie dotąd, dopóki po przeskanowaniu całej planszy nie zostanie dokonana żadna zmiana.
    until unchanged
        unchanged = true # Założenie, że tym razem nie będzie zmian.
        rmin, cmin, pmin = nil # Śledzenie komórki z minimalnym zbiorem możliwości.
        min = 10 # Więcej niż maksymalna liczba możliwości.
        # Iterowanie po komórkach, których wartości są nieznane.
        puzzle.each_unknown do |row, col, box|
            # Znalezienie zbioru wartości, które mogą być w tej komórce.
            p = puzzle.possible(row, col, box)

            # Rozgałęzienie na podstawie rozmiaru zbioru p.
            # Interesują Cię trzy przypadki: p.size==0, p.size==1 i p.size > 1.
            case p.size
            when 0 # Brak możliwości - zbyt surowe ograniczenia.
                raise Impossible
            when 1 # Unikatowa wartość — wstawienie jej do siatki.
                puzzle[row, col] = p[0] # Ustawienie wartości.
                unchanged = false # Zaznaczenie, że dokonano zmiany.
            else # Dla dowolnej innej liczby możliwości.
                # Śledzenie najmniejszego zbioru możliwości.

```

```

# Nie ma problemu, jeśli pętla musi zostać powtórzona.
if unchanged && p.size < min
    min = p.size # Aktualna najmniejsza wartość size.
    rmin, cmin, pmin = row, col, p # Przypisanie równolegle.
end
end
end

# Zwrot komórki z minimalnym zbiorem możliwości.
# Kilka wartości zwrotnych.
return rmin, cmin, pmin
end

# Lamigłówka sudoku powinna być rozwiązywana w miarę możliwości
# przy użyciu prostej logiki. Jeśli zajdzie potrzeba, zostanie zastosowana metoda na siłę.
# Polega ona na użyciu rekursji. Zwraca rozwiązanie
# albo zgłasza wyjątek. Rozwiązywanie jest zwracane w postaci nowego obiektu
# Puzzle bez nieznanych komórek. Metoda ta nie modyfikuje obiektu Puzzle,
# który jest do niej przekazywany. Należy zauważyć, że metoda ta nie potrafi
# wykryć lamigłówków o zbyt mało surowych ograniczeniach.

def Sudoku.solve(puzzle)
    # Tworzenie prywatnej kopii lamigłówki, którą można modyfikować.
    puzzle = puzzle.dup
    # Wstaw tyle liczb, ile jest możliwe za pomocą logiki.
    # Ta metoda modyfikuje lamigłówkę, ale zawsze pozostawia ją nieuszkodzoną.
    # Zwraca numer wiersza, kolumny i zbiór możliwych wartości w tej komórce.
    # Warto zauważać przypisanie równolegle tych wartości zwrotnych do trzech zmiennych.
    r, c, p = scan(puzzle)
    # Jeśli rozwiązanie wyszło przy zastosowaniu logiki, zwróć rozwiązana lamigłówkę.
    return puzzle if r == nil

    # W przeciwnym przypadku spróbuj wstawić każdą z wartości w p dla komórki [r, c].
    # Ponieważ wartości wybierane są ze zbioru możliwych wartości, lamigłówka
    # pozostaje poprawna. Zgadywanie doprowadzi do rozwiązania albo
    # do powstania lamigłówki niemożliwej do rozwiązania. Że lamigłówka nie ma
    # rozwiązania, wiadomo, ponieważ rekursywne wywołanie spowoduje wyjątek. Jeśli tak
    # się stanie, należy zgadywać ponownie lub ponownie wygenerować wyjątek, jeśli
    # wypróbowałeś wszystkie dostępne opcje.
    p.each do |guess|           # Dla każdej wartości w zbiorze możliwych wartości.
        puzzle[r, c] = guess   # Zgadywanie wartości.

        begin
            # Teraz spróbuj (rekursively) rozwiązać zmodyfikowaną lamigłówkę.
            # To rekursywne wywołanie ponownie uruchomi metodę scan() w celu próby rozwiązania
            # zmodyfikowanej lamigłówki przy użyciu logiki.
            # W razie potrzeby nastąpią kolejne próby zgadywania.
            # Pamiętaj, że metoda solve() zwraca rozwiązanie lub
            # generuje wyjątek.
            return solve(puzzle) # Jeśli zwraca wartość, to jest ona rozwiązaniem.
        rescue Impossible
            next                  # Jeśli generuje wyjątek, należy kontynuować zgadywanie.
        end
    end
    # Jeśli dotarłeś do tego miejsca, zgadywanie nic nie dalo.
    # Jedna z wcześniej zgadywanych wartości musiała być złą.
    raise Impossible
end
end

```

Listing 1.1. składa się z 318 wierszy. Ponieważ został on napisany specjalnie z myślą o rozdziale 1., został opatrzony bardzo rozległymi komentarzami. Po usunięciu komentarzy i pustych linijek pozostało 126 wierszy kodu. Jest to bardzo dobry wynik jak na obiektowy program rozwiązywający sudoku nie tylko przy użyciu algorytmu brutalnej siły. Mamy nadzieję, że przykład ten w wystarczającym stopniu demonstruje siłę i ekspresywność języka Ruby.

Struktura i uruchamianie programów Ruby



Niniejszy rozdział opisuje strukturę programów w języku Ruby. Na początku opisano strukturę leksykalną, w tym tokeny i znaki, z których składa się program. Dalej znajduje się opis struktury syntaktycznej programu Ruby z wyjaśnieniem sposobów pisania wyrażeń, instrukcji sterujących, metod, klas itd. jako sekwencji tokenów. Na końcu zamieszczony jest opis plików zawierających kod Ruby z objaśnieniem sposobów dzielenia programów w tym języku na kilka plików oraz wykonywania pliku z kodem Ruby przez interpreter Ruby.

2.1. Struktura leksykalna

Interpreter Ruby analizuje program jako sekwencję **tokenów**. Należą do nich: komentarze, literały, znaki interpunkcyjne, identyfikatory i słowa kluczowe. Niniejszy rozdział wprowadza wymienione typy tokenów oraz zawiera ważne informacje na temat znaków, z których się one składają, w tym rozdzielających białych znaków.

2.1.1. Komentarze

Komentarze zaczynają się od znaku # i mają zasięg do końca wiersza. Interpreter Ruby ignoriuje znaki # i wszystko, co znajduje się za nimi w tym samym wierszu (nie ignoruje znaku nowego wiersza, który jest białym znakiem mogącym oznaczać zakończenie instrukcji).

Jeśli # znajduje się wewnętrz łańcucha lub wyrażenia regularnego (zobacz rozdział 3.), to stanowi jego część i nie wprowadza komentarza.

```
# Ten cały wiersz jest komentarzem.  
x = "#To jest łańcuch."          # A to jest komentarz.  
y = /#To jest wyrażenie regularne./ # Tu jest jeszcze jeden komentarz.
```

Komentarze zajmujące wiele wierszy są tworzone przez wstawienie znaku # na początku każdego z nich.

```
#  
# Niniejsza klasa reprezentuje liczbę typu Complex.  
# Mimo nazwy (złożona) nie jest ona wcale skomplikowana.  
#
```

Należy zauważyć, że w języku Ruby nie ma komentarzy w stylu C /* */. Nie ma sposobu na wstawienie komentarza w środku wiersza kodu.

2.1.1.1. Dokumenty osadzone

W języku Ruby dostępny jest jeszcze inny sposób wstawiania komentarzy wielowierszowych. Są to tak zwane **dokumenty osadzone** (ang. *embedded document*). Pierwszy wiersz takiego komentarza zaczyna się od ciągu znaków =begin, a ostatni od ciągu =end. Tekst znajdujący się pomiędzy =begin a =end jest ignorowany. Należy tylko pamiętać, że między tekstem a ciągami =begin i =end musi być przynajmniej jedna spacja.

Dokumenty osadzone są wygodnym sposobem na tworzenie długich komentarzy bez wstawiania na początku każdego wiersza znaku #.

```
=begin Ktoś musi naprawić poniższy kod!  
Kod znajdujący się w tym miejscu jest w komentarzu.  
=end
```

Warto zauważyć, że dokumenty osadzone działają tylko wówczas, gdy wiersze zaczynają się od znaku =.

```
# =begin Ten wiersz był komentarzem, a teraz sam jest w komentarzu!
  Kod znajdujący się w tym miejscu nie jest w komentarzu.
# =end
```

Jak sama nazwa wskazuje, dokumenty osadzone mogą służyć do wstawiania w programie Ruby długich bloków dokumentacji lub kodu źródłowego w innym języku (na przykład HTML lub SQL). Są one często przeznaczone do użytku przez różnego rodzaju narzędzia przetwarzania końcowego, które są uruchamiane na rzecz kodu Ruby. Po ciągu =begin z reguły umieszczany jest identyfikator określający, dla jakiego narzędzia przeznaczony jest dany komentarz.

2.1.1.2. Komentarze dokumentacyjne

W programach Ruby można osadzać dokumentację API w specjalnych, przeznaczonych do tego celu komentarzach poprzedzających definicje metod, klas i modułów. Do przeglądania tej dokumentacji służy omówione w rozdziale 1.2.4 narzędzie o nazwie ri. Narzędzie rdoc pobiera komentarze dokumentacyjne z kodu źródłowego Ruby i konwertuje je na format HTML lub przygotowuje do wyświetlenia przez narzędzie ri. Opis narzędzia rdoc wykracza poza tematykę tej książki. Szczegółowe informacje na jego temat znajdują się w pliku *lib/rdoc/README* w kodzie źródłowym Ruby.

Komentarz dokumentacyjny musi znajdować się bezpośrednio przed modelem, klasą lub metodą, które API dokumentuje. Z reguły zajmuje kilka wierszy zaczynających się od znaku #, ale może też mieć formę dokumentu osadzonego rozpoczynającego się ciągiem =begin rdoc (jeśli słowo rdoc zostanie pominięte, narzędzie rdoc nie zauważycie tego komentarza).

Poniższy komentarz demonstruje najważniejsze elementy formatujące gramatyki oznaczania komentarzy dokumentacyjnych w języku Ruby. Szczegółowy opis tej gramatyki znajduje się w wymienionym wcześniej pliku *README*:

```
#  
# Gramatyka oznaczania komentarzy rdoc jest prosta jak w wiki.  
#  
# Akapity oddziela się pustym wierszem.  
#  
# = Nagłówki.  
#  
# Nagłówki zaczynają się od znaku równości.  
#  
# == Podnagłówki.  
# Powyższy wiersz tworzy podnagłówek.  
# === Podpodnagłówek.  
# I tak dalej.  
#  
# = Przykłady.  
#  
# Wcięte wiersze są wyświetlane dosłownie pismem o stałej szerokości znaków.  
# Należy uważać, aby nie wciąć nagłówków lub list.  
#  
# = Listy i rodzaje pisma.  
#  
# Elementy listy zaczynają się od znaku * lub -. Rodzaj pisma określa się za pomocą interpunkcji lub kodu HTML:  
# * italic lub <i>kursywa</i>  
# * bold lub <b>pogrubienie</b>  
# * +code+ lub <tt>pismo o stałej szerokości znaków</tt>  
#
```

```

# 1. Listy numerowane zaczynają się od liczb.
# 99. Można używać dowolnych liczb; nie muszą być kolejne.
# 1. Nie ma sposobu na zagnieżdżanie list.
#
# Terminy list opisowych są umieszczane w nawiasach kwadratowych:
# [element 1] Opis elementu 1.
# [element 2] Opis elementu 2.
#

```

2.1.2. Literały

Literały to wartości, które znajdują się bezpośrednio w kodzie źródłowym Ruby. Zaliczają się do nich liczby,łańcuchy tekstowe i wyrażenia regularne (inne literały, takie jak wartości tablic jednowymiarowych i asocjacyjnych, nie są pojedynczymi tokenami, a bardziej złożonymi wyrażeniami). Składnia literałów liczbowych i łańcuchowych w języku Ruby jest skomplikowana, a została szczegółowo opisana w rozdziale 3. Na razie wystarczy tylko krótki przykład demonstrujący, jak wyglądają literały w języku Ruby:

1	# Literal całkowitaliczbowy.
1.0	# Literal liczby zmienoprzecinkowej.
'one'	# Literal łańcuchowy.
"two"	# Inny literal łańcuchowy.
/trzy/	# Literal wyrażenia regularnego.

2.1.3. Znaki interpunkcyjne

Znaki interpunkcyjne spełniają w języku Ruby różne zadania. Ich postać ma większość operatorów w tym języku, na przykład operator dodawania to +, mnożenia *, a logiczne LUB to ||. Pełna lista operatorów w języku Ruby znajduje się w rozdziale 4.6. Znaki specjalne służą także do oddzielania łańcuchów, wyrażeń regularnych, literałów tablic jednowymiarowych i asocjacyjnych oraz grupowania wyrażeń, argumentów metod oraz indeksów tablic. W składni języka Ruby znaki interpunkcyjne mają także wiele innych zastosowań, o których będzie jeszcze mowa.

2.1.4. Identyfikatory

Identyfikator to inaczej nazwa. W języku Ruby wykorzystuje się go do nazywania zmiennych, metod, klas itd. Może składać się z liter, liczb i znaków podkreślenia, ale nie może zaczynać się od liczby. Nie może zawierać znaków: białych, niedrukowalnych oraz interpunkcyjnych z wyjątkiem wymienionych tutaj.

Identyfikatory zaczynające się od wielkiej litery A-Z oznaczają stałe. Jeśli wartość takiego identyfikatora zostanie zmieniona w programie, interpreter zgłosi ostrzeżenie (ale nie błąd). Nazwy klas i modułów muszą zaczynać się od wielkiej litery. Poniżej znajduje się kilka przykładowych identyfikatorów:

```

i
x2
old_value
internal      # Identyfikatory mogą zaczynać się od znaku podkreślenia.
PI             # Stała.

```

Zgodnie z konwencją identyfikatory składające się z kilku słów niebędących stałymi pisane są ze znakiem podkreślenia `w_taki_sposób`, podczas gdy identyfikatory stałych, które składają się z kilku wyrazów, pisze się `WTakiSposób` lub `W_TAKI_SPOSÓB`.

2.1.4.1. Rozróżnianie wielkich i małych liter

Wielkość liter w języku Ruby ma znaczenie. Mała litera i wielka litera to nie to samo. Na przykład słowo kluczowe `end` jest czymś innym niż słowo kluczowe `END`.

2.1.4.2. Znaki Unicode w identyfikatorach

Reguły języka Ruby dotyczące tworzenia identyfikatorów są zdefiniowane w kategoriach znaków ASCII, które są zabronione. Ogólnie mówiąc, wszystkie znaki spoza zestawu ASCII mogą być używane w identyfikatorach, wliczając te wyglądające na znaki interpunkcyjne. Na przykład w pliku UTF-8 poniższy kod Ruby jest poprawny:

```
def *(x,y) # Nazwa tej metody jest znak mnożenia Unicode.  
    x*y      # Metoda ta mnoży podane argumenty.  
end
```

Podobnie japoński programista może w swoim programie zakodowanym w systemie SJIS lub EUC używać w identyfikatorach znaków Kanji.Więcej informacji na temat pisania programów kodowanych w innych systemach niż ASCII znajduje się w podrozdziale 2.4.1.

Specjalne reguły dotyczące tworzenia identyfikatorów są oparte na znakach ASCII i nie dotyczą znaków spoza tego zestawu. Na przykład identyfikator nie może zaczynać się od cyfry z zestawu ASCII, ale może zaczynać się od cyfry z alfabetu innego niż typu Latin. Podobnie, aby być stałą, musi zaczynać się od wielkiej litery z zestawu ASCII. Na przykład identyfikator `Ł` nie jest stałą.

Dwa identyfikatory są takie same tylko wówczas, gdy są reprezentowane przez taki sam ciąg bajtów. Niektóre zestawy znaków, na przykład Unicode, posiadają więcej niż jeden punkt kodowy do reprezentacji tego samego znaku. W języku Ruby nie ma żadnej normalizacji Unicode, a więc różne punkty kodowe są uznawane za różne znaki, nawet jeśli mają identyczne znaczenie lub są reprezentowane przez taki sam glif czcionki.

2.1.4.3. Znaki interpunkcyjne w identyfikatorach

Znaki interpunkcyjne mogą występować na początku i końcu identyfikatorów. Ich znaczenie jest następujące:

- \$ Znak ten przed nazwą zmiennej oznacza, że jest ona globalna. Podążając za przykładem Perla, w języku Ruby istnieje grupa zmiennych globalnych, które w nazwach zawierają inne znaki interpunkcyjne, takie jak `$_` i `$-`. Lista tych specjalnych zmiennych znajduje się w rozdziale 10.
- @ Na początku zmiennych obiektowych znajduje się symbol `@`. Zmienne klasowe mają przedrostek złożony z dwóch takich znaków. Zmienne obiektowe i klasowe zostały opisane w rozdziale 7.
- ? Bardzo pomocną konwencją jest umieszczanie na końcu nazw metod zwracających wartości logiczne znaku `?`.
- ! Nazwy metod kończące się znakiem wykrywki wskazują, że należy ostrożnie ich używać. Konwencja ta jest z reguły stosowana do wyróżnienia metod mutacyjnych modyfikujących obiekty bezpośrednio w porównaniu z metodami, które zwracają zmodyfikowaną kopię obiektu.
- = Metody posiadające nazwy kończące się znakiem równości można wywoływać po przez wstawienie ich nazw bez znaku równości po lewej stronie operatora przypisania (więcej na ten temat można przeczytać w podrozdziałach 4.5.3 i 7.1.5).

Poniżej znajduje się kilka identyfikatorów z przedrostkami lub przyrostkami:

```
$files          # Zmienna globalna.  
@data          # Zmienna obiektowa.  
@@counter      # Zmienna klasowa.  
empty?         # Metoda zwracająca wartość logiczną, czyli predykat.  
sort!          # Wersja metody sort modyfikująca obiekty bezpośrednio.  
timeout=       # Metoda wywoływana przez przypisanie.
```

Niektóre operatory języka Ruby są zaimplementowane jako metody, dzięki czemu można je przedefiniowywać w różnych klasach zgodnie z potrzebą. Dlatego też istnieje możliwość używania niektórych operatorów jako nazw metod. W tym kontekście znaki interpunkcyjne lub znaki operatorów są traktowane jako identyfikatory, a nie operatory. Więcej informacji na temat operatorów w języku Ruby znajduje się w podrozdziale 4.6.

2.1.5. Słowa kluczowe

Poniższe słowa kluczowe mają w języku Ruby specjalną funkcję i tak też są traktowane przez analizator składni Ruby:

__LINE__	case	ensure	not	then
__ENCODING__	class	false	or	true
__FILE__	def	for	redo	undef
BEGIN	defined?	if	rescue	unless
END	do	in	retry	until
alias	else	module	return	when
and	elsif	next	self	while
begin	end	nil	super	yield
break				

Poza powyższymi słowami istnieją jeszcze trzy tokeny przypominające słowa kluczowe. Są one specjalnie traktowane przez analizator składni tylko wówczas, gdy znajdują się na początku wiersza.

```
=begin      =end      __END__
```

Jak wiadomo, tokeny `=begin` i `=end` na początku wiersza oznaczają początek i koniec komentara wielowierszowego. Token `__END__` oznacza koniec programu (i początek sekcji danych), jeśli znajduje się sam w wierszu (ani przed nim, ani za nim nie może być żadnych białych znaków).

W większości języków słowa kluczowe byłyby tak zwanyimi słowami zarezerwowanymi i nigdy nie można by było używać ich jako identyfikatorów. Analizator składni Ruby jest bardzo elastyczny. Nie zgłasza żadnych błędów, jeśli któreś z tych słów zostanie opatrzone przedrostkiem `@`, `@@` czy `$` i użyte jako identyfikator zmiennej obiektowej, klasowej lub globalnej. Ponadto słów kluczowych można używać jako nazw metod, z tym, że metoda musi zawsze być wywoływana jawnie poprzez obiekt. Należy jednak pamiętać, że użycie tych słów jako identyfikatorów spowoduje zaciemnienie kodu źródłowego. Najlepiej jest traktować je jako słowa zarezerwowane.

Wiele ważnych własności języka Ruby jest zaimplementowanych jako metody klas `Kernel`, `Module`, `Class` i `Object`. Dlatego też dobrze jest poniższe słowa również traktować jako zarezerwowane:

```
# To są metody, które wyglądają na instrukcje lub słowa kluczowe.  
at_exit        catch        private      require  
attr           include      proc         throw  
attr_accessor  lambda      protected   public  
attr_reader    load        raise
```

To są często używane funkcje globalne.

Array	chomp!	gsub!	select
Float	chop	iterator?	sleep
Integer	chop!	load	split
String	eval	open	sprintf
URI	exec	p	srand
abort	exit	print	sub
autoload	exit!	printf	sub!
autoload?	fail	putc	syscall
binding	fork	puts	system
block_given?	format	rand	test
callcc	getc	readline	trap
caller	gets	readlines	warn
chomp	gsub	scan	

To są często używane metody obiektowe.

allocate	freeze	kind_of?	superclass
clone	frozen?	method	taint
display	hash	methods	tainted?
dup	id	new	to_a
enum_for	inherited	nil?	to_enum
eql?	inspect	object_id	to_s
equal?	instance_of?	respond_to?	untaint
extend	is_a?	send	

2.1.6. Białe znaki

Spacje, tabulatory i znaki nowego wiersza nie są same w sobie tokenami, ale służą do ich oddzielania, gdyż te w przeciwnym razie zlałyby się w jeden. Poza tą podstawową funkcją rozdzielającą większość białych znaków jest ignorowana przez interpreter Ruby. Są one stosowane głównie do formatowania programów, aby łatwiej się je czytało. Jednak nie wszystkie białe znaki są ignorowane. Niektóre są wymagane, a niektóre nawet zabronione. Gramatyka języka Ruby jest ekspresywna, ale i skomplikowana. Istnieje kilka przypadków, w których wstawienie lub usunięcie białego znaku może spowodować zmianę działania programu. Mimo iż sytuacje takie nie są częste, należy o nich wiedzieć.

2.1.6.1. Znaki nowego wiersza jako znaki kończące instrukcje

Najczęstsze zastosowanie białych znaków ma związek ze znakami nowego wiersza, które służą jako znaki kończące instrukcje. W językach takich jak Java i C każda instrukcja musi być zakończona średnikiem. W języku Ruby także można kończyć instrukcje tym znakiem, ale jest to wymagane tylko wówczas, gdy w jednym wierszu znajduje się więcej niż jedna instrukcja. Zgodnie z konwencją we wszystkich innych tego typu sytuacjach średnik nie jest używany.

Przy braku średników interpreter Ruby musi zgadnąć na własną rękę, gdzie kończy się dana instrukcja. Jeśli kod w wierszu stanowi pełną pod względem syntaktycznym instrukcję, znakiem ją kończącym jest znak nowego wiersza. Gdy instrukcja nie jest kompletna, Ruby kontynuuje jej analizę w kolejnym wierszu (w Ruby 1.9 jest jeden wyjątek od tej reguły, który został opisany dalej w tym podrozdziale).

Jeżeli wszystkie instrukcje mieszą się w pojedynczym wierszu, nie ma problemu. Jednak w przypadku gdy instrukcja zajmuje więcej niż jeden wiersz, należy tak ją podzielić, aby interpreter nie wziął jej pierwszej części za kompletną instrukcję. W takiej sytuacji w grę wchodzi biały znak. Program może działać na różne sposoby w zależności od tego, gdzie się on znajduje. Na przykład poniższa procedura dodaje x do y i sumę zapisuje w zmiennej total:

```
total = x +      # Wyrażenie niekompletne — analiza jest kontynuowana.  
y
```

Natomiast poniższy kod przypisuje zmienną `x` do `total`, a następnie oblicza wartość `y` i nic z nią nie robi:

```
total = x  # To jest kompletne wyrażenie.  
+ y       # Bezużyteczne, ale kompletne wyrażenie.
```

Kolejnym przykładem są instrukcje `return` i `break`, po których opcjonalnie może znajdować się wyrażenie określające wartość zwrotną. Znak nowego wiersza pomiędzy słowem kluczowym a wyrażeniem spowoduje zakończenie instrukcji przed tym ostatnim.

Znak nowego wiersza można wstawić bez obawy, że instrukcja zostanie zakończona zbyt wcześnie, po każdym operatorze, kropce lub przecinku w wywołaniu metody oraz literale dowolnego rodzaju tablicy.

Można go także zastąpić lewym ukośnikiem, co zapobiega automatycznemu zakończeniu instrukcji przez Ruby:

```
var total = first_long_variable_name + second_long_variable_name \  
+ third_long_variable_name # Powyżej nie ma żadnego znaku kończącego instrukcję.
```

Reguły dotyczące zakończenia instrukcji są nieco inne w Ruby 1.9. Jeśli pierwszym znakiem niebędącym spacją w wierszu jest kropka, wiersz ten jest traktowany jako kontynuacja poprzedniego, a więc znajdujący się wcześniej znak nowego wiersza nie kończy instrukcji. Wiersze zaczynające się od kropki są przydatne w przypadku długich łańcuchów metod, czasami nazywanych płynnymi API (ang. *fluent API*); każda wywołana w nich metoda zwraca obiekt, na rzecz którego mogą być wywołane dodatkowe metody. Na przykład:

```
animals = Array.new  
.push("pies")  # Nie działa w Ruby 1.8.  
.push("krowa")  
.push("kot")  
.sort
```

2.1.6.2. Spacje a wywoływanie metod

W niektórych sytuacjach gramatyka Ruby dopuszcza pominięcie nawiasów w wywołaniach metod, które dzięki temu mogą być używane tak, jakby były instrukcjami; w znacznym stopniu wpływa to na elegancję kodu. Niestety, możliwość ta powoduje zależność od białych znaków. Przykładowo poniższe dwa wiersze kodu różnią się tylko jedną spacją:

```
f(3+2)+1  
f (3+2)+1
```

W pierwszym wierszu do funkcji `f` zostaje przekazana wartość 5, a do zwróconego wyniku zostaje dodana jedynka. Ponieważ w drugim wierszu po nazwie funkcji znajduje się spacja, Ruby zakłada, że nawias w wywołaniu tej funkcji został pominięty. Nawias znajdujący się dalej jest traktowany jako sposób oddzielenia wyrażenia, a argumentem funkcji jest całe wyrażenie $(3+2)+1$. Jeśli włączone są ostrzeżenia (za pomocą opcji `-w`), Ruby zgłasza ostrzeżenie `wawsze`, gdy napotyka taki niejednoznaczny kod.

Rozwiązywanie tego problemu jest proste:

- Nigdy nie należy umieszczać spacji między nazwą metody a otwierającym nawiasem.
- Jeżeli pierwszy argument metody zaczyna się od nawiasu otwierającego, wywołanie metody zawsze powinno być otoczone nawiasami. Na przykład `f((3+2)+1)`.

- Zawsze uruchamiaj interpreter z opcją `-w`, dzięki czemu będzie on zgłaszał ostrzeżenia zawsze, gdy zapomnisz o powyższych regułach!

2.2. Struktura syntaktyczna

Do tej pory zostały omówione tokeny i znaki, z których się one składają. Teraz krótko zajmiemy się tym, jak tokeny leksykalne łączą się w większe struktury syntaktyczne programu Ruby. Niniejszy podrozdział opisuje składnię programów Ruby od najprostszych wyrażeń po największe moduły. W efekcie jest on mapą prowadzącą do kolejnych rozdziałów.

Podstawową jednostką syntaktyczną w języku Ruby jest **wyrażenie** (ang. *expression*). Interpreter Ruby **oblicza** wyrażenia, zwracając ich wartości. Najprostsze wyrażenia to **wyrażenia pierwotne** (ang. *primary expression*), które bezpośrednio reprezentują wartości. Należą do nich opisywane wcześniej literały łańcuchowe i liczbowe. Inne tego typu wyrażenia to niektóre słowa kluczowe, jak `true`, `false`, `nil` i `self`. Odwołania do zmiennych również są wyrażeniami pierwotnymi. Ich wartością jest wartość zmiennej.

Bardziej skomplikowane wartości mogą być zapisane jako wyrażenia złożone:

```
[1,2,3]          # Literal tablicowy.
{1=>"one", 2=>"two"}  # Literal tablicy asocjacyjnej.
1..3            # Literal zakresowy.
```

Operatory służą do wykonywania obliczeń na wartościach, a wyrażenia złożone zbudowane są z prostszych podwyrażeń rozdzielonych operatorami:

```
1           # Wyrażenie pierwotne.
x           # Inne wyrażenie pierwotne.
x = 1       # Wyrażenie przypisania.
x = x + 1  # Wyrażenie z dwoma operatorami.
```

Tematowi operatorów i wyrażeń, w tym zmiennych i wyrażeń przypisania, poświęcony jest rozdział 4.

Wyrażenia w połączeniu ze słowami kluczowymi tworzą instrukcje, jak na przykład instrukcja `if`, która warunkowo wykonuje kod, lub instrukcja `while` wykonująca kod wielokrotnie:

```
if x < 10 then  # Jeśli to wyrażenie ma wartość true,
  x = x + 1    # należy wykonać tę instrukcję.
end             # Oznacza koniec instrukcji warunkowej.
while x < 10 do  # Dopóki wyrażenie to ma wartość true...
  print x      # należy wykonywać tę instrukcję.
  x = x + 1    # Następnie należy wykonać tę instrukcję.
end             # Oznacza koniec pętli.
```

W języku Ruby instrukcje te są z technicznego punktu widzenia wyrażeniami, ale nadal istnieje przydatne rozróżnienie pomiędzy wyrażeniami na wpływające na przepływ sterowania w programie i na te, które tego nie robią. Instrukcje sterujące zostały opisane w rozdziale 5.

We wszystkich nieprymitywnych programach wyrażenia i instrukcje grupowane są w parametryzowane jednostki, dzięki czemu mogą być wielokrotnie wywoływanie przy użyciu różnych danych wejściowych. Jednostki te są nazywane funkcjami, procedurami lub podprocedurami. Ponieważ język Ruby jest zorientowany obiektowo, jednostki te to **metody**. Metody i związane z nimi struktury nazywane `proc` i `lambda` są tematem rozdziału 6.

Zestawy metod zaprojektowane, aby wzajemnie oddziaływały między sobą, można łączyć w klasy, a grupy wzajemnie powiązanych klas i metod, które są od nich niezależne, tworzą **moduły**. Klassy i moduły są tematem rozdziału 7.

2.2.1. Struktura bloku

Programy Ruby mają strukturę blokową. Moduły, klasy i definicje metod, a także większość instrukcji zawiera bloki zagnieździonego kodu. Są one oznaczane słowami kluczowymi lub specjalnymi znakami i zgodnie z konwencją powinny być wcięte na głębokość dwóch spacji względem swoich ograniczników. W programach w języku Ruby mogą występować dwa rodzaje bloków. Jeden z nich jest formalnie nazywany blokiem. Jest to fragment kodu związany z metodą iteracyjną lub do niej przekazywany:

```
3.times { print "Ruby! " }
```

W powyższym przykładzie klamry wraz ze znajdującym się między nimi kodem stanowią blok związany z wywołaniem metody iteracyjnej `3.times`. Formalne bloki tego typu mogą być oznaczane klamrami lub słowami kluczowymi `do` i `end`:

```
1.upto(10) do |x|
  print x
end
```

Ograniczniki `do` i `end` są z reguły używane w przypadkach, gdy blok zajmuje więcej niż jeden wiersz kodu. Zwróć uwagę na wcięcie wielkości dwóch spacji kodu w bloku. Bloki zostały opisane w podrozdziale 5.4.

Aby uniknąć mylenia prawdziwych bloków, drugi ich rodzaj można nazwać **ciałem** (w praktyce jednak termin „blok” jest używany w obu przypadkach). Ciało jest listą instrukcji, które składają się na ciało definicji klasy, metody, pętli `while` lub czegokolwiek innego. Nigdy nie jest oznaczane klamrami — w tym przypadku ogranicznikami są słowa kluczowe. Szczegóły dotyczące składni ciał instrukcji, metod oraz klas i modułów znajdują się odpowiednio w rozdziałach 5., 6. i 7.

Ciała i bloki można zagnieźdzać jedne w drugich. Programy języka Ruby zazwyczaj zawierają kilka poziomów zagnieździonego kodu, czytelnego dzięki wcięciom. Poniżej znajduje się schematyczny przykład:

```
module Stats
  class Dataset
    # Moduł.
    def initialize(filename)
      # Klasa w module.
      IO.foreach(filename) do |line|
        # Metoda w klasie.
        if line[0,1] == "#"
          # Blok w metodzie.
          next
          # Instrukcja if w bloku.
        end
        # Prosta instrukcja w instrukcji if.
      end
      # Koniec ciała instrukcji if.
    end
    # Koniec ciała metody.
  end
  # Koniec ciała klasy.
end
# Koniec ciała modułu.
```

2.3. Struktura plików

Zasad dotyczących struktury kodu języka Ruby w pliku jest kilka. Dotyczą one przygotowywania programów do użytku i nie dotyczą bezpośrednio samego języka.

Po pierwsze, jeśli w programie Ruby zawarty jest komentarz shebang (`#!/`) informujący systemy operacyjne typu Unix, jak go uruchomić, musi on znajdować się w pierwszej linijce.

Po drugie, w sytuacji gdy w programie Ruby znajduje się komentarz określający kodowanie znaków (opisane w podrozdziale 2.4.1), musi on znajdować się w pierwszej linijce lub w drugiej, jeśli w pierwszej jest komentarz shebang.

Po trzecie, jeżeli plik zawiera linijkę, w której znajduje się tylko token `_END_` bez żadnych białych znaków przed nim i za nim, interpreter kończy przetwarzanie w tym miejscu. W dalszej części pliku mogą znajdować się dowolne dane, które program może odczytywać za pomocą stałej `DATA` obiektu `IO` (więcej informacji na temat tej stałej globalnej można znaleźć w podrzędziale 9.7 i rozdziale 10.).

Program Ruby nie musi mieścić się w jednym pliku. Na przykład wiele programów ładuje kod Ruby z dodatkowych plików bibliotecznych. Do ładowania kodu z innych plików służy metoda `require`. Szuka ona określonych modułów na ścieżce wyszukiwania i uniemożliwia załadowanie danego modułu więcej niż jeden raz. Szczegóły na ten temat znajdują się w podrzędziale 7.6.

Poniższy kod ilustruje każdy z wymienionych punktów struktury pliku z programem Ruby:

<code>#!/usr/bin/ruby -w</code>	<i>Komentarz shebang.</i>
<code># -*- coding: utf-8 -*-</code>	<i>Komentarz określający kodowanie.</i>
<code>require 'socket'</code>	<i>Załadowanie biblioteki sieciowej.</i>
<code>...</code>	<i>Kod programu.</i>
<code>__END__</code>	<i>Koniec programu.</i>
<code>...</code>	<i>Dane programu.</i>

2.4. Kodowanie znaków

Na najniższym poziomie program w języku Ruby jest ciągiem znaków. Reguły leksykalne tego języka zostały zdefiniowane przy użyciu znaków z zestawu ASCII. Na przykład komentarze zaczynają się od znaku `#` (kod ASCII 35), a dozwolone białe znaki to: tabulator poziomy (ASCII 9), znak nowego wiersza (10), tabulator pionowy (11), wysuw strony (12), powrót karetki (13) i spacja (32). Wszystkie słowa kluczowe języka Ruby zostały zapisane znakami ASCII; także wszystkie operatory i inne znaki interpunkcyjne pochodzą z tego zestawu.

Domyślnie interpreter Ruby przyjmuje, że kod źródłowy Ruby jest zakodowany w systemie ASCII. Nie jest to jednak wymóg. Interpreter może przetwarzać także pliki zakodowane w innych systemach zawierających wszystkie znaki dostępne w ASCII. Aby mógł on zinterpretować bajty pliku źródłowego jako znaki, musi wiedzieć, jakiego kodowania użyć. Kodowanie mogą określać pliki Ruby lub można poinformować o tym interpreter. Wyjaśnienie, jak to zrobić, znajduje się nieco dalej.

Interpreter Ruby jest bardzo elastyczny, jeśli chodzi o znaki występujące w programach. Niektóre znaki ASCII mają specjalne znaczenie i nie mogą być stosowane w identyfikatorach. Poza tym program Ruby może zawierać wszelkie znaki dozwolone przez kodowanie. Napisaliśmy wcześniej, że identyfikatory mogą zawierać znaki spoza zestawu ASCII. To samo dotyczy komentarzy, literałów łańcuchowych i wyrażeń regularnych — mogą zawierać dowolne znaki inne niż znak ograniczający oznaczający koniec komentarza lub literała. W plikach ASCII łańcuchy mogą zawierać dowolne bajty, także te, które reprezentują niedrukowalne znaki kontrolne (takie użycie surowych bajtów nie jest jednak zalecane; w literałach w języku Ruby można stosować symbole zastępcze, dzięki czemu dowolne znaki można wstawić przy

użyciu kodów liczbowych). Jeśli plik jest zakodowany w systemie UTF-8, komentarze, łańcuchy i wyrażenia regularne mogą zawierać dowolne znaki Unicode. Jeśli plik jest zakodowany w jednym z japońskich systemów — SJIS lub EUC — łańcuchy mogą zawierać znaki Kanji.

2.4.1. Określanie kodowania programu

Domyślnie interpreter Ruby przyjmuje, że programy są kodowane w systemie ASCII. W Ruby 1.8 kodowanie można zmienić za pomocą opcji wiersza poleceń `-K`. Aby uruchomić program Ruby zawierający znaki Unicode w UTF-8, należy uruchomić interpreter przy użyciu opcji `-Ku`. Programy zawierające japońskie znaki w kodowaniu EUC-JP lub SJIS można uruchomić, wykorzystując opcję `-Ke` i `-Ks`.

Ruby 1.9 również obsługuje opcję `-K`, ale nie jest ona już preferowanym sposobem określania kodowania pliku z programem. Zamiast zmuszać użytkownika skryptu do określania kodowania w trakcie uruchamiania Ruby, twórca skryptu może je określić za pomocą specjalnego komentarza znajdującego się na początku pliku¹. Na przykład:

```
# coding: utf-8
```

Komentarz musi składać się wyłącznie ze znaków ASCII i zawierać słowo `coding` z dwukropkiem lub znakiem równości, po którym znajduje się nazwa wybranego kodowania (nie może ona zawierać spacji ani znaków interpunkcyjnych z wyjątkiem myślnika i znaku podkreślenia). Białe znaki mogą znajdować się po obu stronach dwukropka lub znaku równości, a przed łańcuchem `coding` może znajdować się dowolny przedrostek, jak `en`. W całym tym komentarzu, wyłącznie ze słowem `coding` i nazwą kodowania, nie są rozróżniane wielkie i małe litery, a więc może on być pisany zarówno małymi, jak i wielkimi literami.

Komentarze kodowania zazwyczaj zawierają także informację o kodowaniu dla edytora tekstuowego. Użytkownicy edytora Emacs mogliby napisać:

```
# -*- coding: utf-8 -*-
```

A użytkownicy programu vi:

```
# vi: set fileencoding=utf-8 :
```

Tego typu komentarz kodowania zazwyczaj może znajdować się tylko w pierwszej linijce pliku. Wyjątkiem jest sytuacja, gdy pierwsza linijka jest zajęta przez komentarz shebang (który umożliwia wykonanie skryptu w systemach uniksowych). Wówczas kodowanie może znajdować się w drugiej linijce.

```
#!/usr/bin/ruby -w  
# coding: utf-8
```

W nazwach kodowania nie są rozróżniane wielkie i małe litery, a więc nazwy można pisać w dowolny sposób, także mieszając małe litery z wielkimi. Ruby 1.9 obsługuje następujące kodowania źródła: ASCII-8BIT (inna nazwa to BINARY), US-ASCII (7-bit ASCII), kodowania europejskie ISO-8859-1 do ISO-8859-15, Unicode UTF-8 oraz japońskie SHIFT_JIS (inaczej SJIS) i EUC-JP. Konkretne komplikacje lub dystrybucje Ruby mogą obsługiwać także dodatkowe kodowania.

Pliki zakodowane w systemie UTF-8 identyfikują swoje kodowanie, jeśli ich trzy pierwsze bajty to 0xEF 0xBB 0xBF. Bajty te nazywane są BOM (*Byte Order Mark* — znacznik kolejności bajtów) i nie są obowiązkowe w plikach UTF-8 (niektóre programy działające w systemie Windows dodają te bajty przy zapisywaniu plików Unicode).

¹ W tej kwestii Ruby wykorzystuje konwencję z języka Python; zobacz <http://www.python.org/dev/peps/pep-0263/>.

W Ruby 1.9 słowo kluczowe `_ENCODING_` (na początku i końcu znajdują się dwa znaki podkreślenia) ewaluuje do kodowania źródła aktualnie wykonywanego kodu. Powstała w wyniku tego wartość jest obiektem klasy `Encoding` (więcej na temat klasy `Encoding` znajduje się w podrozdziale 3.2.6.2).

2.4.2. Kodowanie źródła i domyślne kodowanie zewnętrzne

W Ruby 1.9 ważna jest różnica pomiędzy kodowaniem źródła pliku Ruby a **domyślnym kodowaniem zewnętrznym** procesu Ruby. Kodowanie źródła jest tym, co zostało opisane wcześniej — informuje interpreter, jak odczytywać znaki w skrypcie. Jest ono zazwyczaj ustawiane za pomocą komentarzy kodowania. Program Ruby może składać się z więcej niż jednego pliku, a każdy z nich może mieć inne kodowanie źródła. Kodowanie źródła pliku wpływa na kodowanie literalów łańcuchowych w tym pliku. Więcej informacji na temat kodowania łańcuchów znajduje się w podrozdziale 3.2.6.

Domyślne kodowanie zewnętrzne to coś innego — jest ono używane przez Ruby podczas odczytu z plików i strumieni. Obejmuje cały proces Ruby i nie może zmieniać się od pliku do pliku. W typowych warunkach jest ustawiane na podstawie lokalizacji, na którą ustawiony jest komputer. Można to jednak zmienić, używając odpowiednich opcji wiersza poleceń, o czym za chwilę. Domyślne kodowanie zewnętrzne nie wpływa na kodowanie literalów łańcuchowych, ale ma duże znaczenie dla operacji wejścia i wyjścia; więcej informacji na ten temat znajduje się w podrozdziale 9.7.2.

Wcześniej podana została opcja `-K` jako sposób na określenie kodowania źródła. W rzeczywistości ustawia ona domyślne kodowanie zewnętrzne procesu i stosuje je jako domyślne kodowanie źródła.

W Ruby 1.9 opcja `-K` jest dostępna ze względu na zgodność z Ruby 1.8, ale nie jest już zalecanym sposobem ustawiania kodowania zewnętrznego. Dwie nowe opcje `-E` i `--encoding` pozwalają na określenie kodowania za pomocą jego pełnej nazwy zamiast jednoliterowego skrótu. Na przykład:

```
ruby -E utf-8          # Nazwa kodowania po opcji -E.  
ruby -Eutf-8          # Spacja jest opcjonalna.  
ruby --encoding utf-8 # Nazwa kodowania po opcji --encoding ze spacją.  
ruby --encoding=utf-8 # Po opcji --encoding można wstawić znak równości.
```

Wszystkie szczegóły na ten temat znajdują się w podrozdziale 10.1.

Domyślne kodowanie zewnętrzne można sprawdzić, wykorzystując klasową metodę `Encoding.default_external`. Zwraca ona obiekt typu `Encoding`. Aby sprawdzić nazwę (jako łańcuch) kodowania znaków pochodzącego od lokalizacji, należy użyć metody `Encoding.locale_encoding`; zawsze bazuje ona na ustawieniach dotyczących lokalizacji i ignoruje opcje wiersza poleceń zmieniające domyślne kodowanie zewnętrzne.

2.5. Wykonywanie programu

Ruby to język skryptowy. Oznacza to, że programy Ruby są listami lub skryptami poleceń do wykonania. Domyślnie polecenia są wykonywane sekwencyjnie w takiej kolejności, w jakiej zostały napisane. Instrukcje sterujące (opisane w rozdziale 5.) zmieniają tę domyślną kolejność, pozwalając na warunkowe wykonywanie niektórych instrukcji lub wielokrotne ich powtarzanie.

Programiści przyzwyczajeni do tradycyjnych statycznych języków kompilowanych, jak C lub Java, mogą być nieco zbici z tropu. W Ruby nie ma specjalnej metody `main`, od której zaczyna się wykonywanie. Interpreter Ruby otrzymuje skrypt instrukcji do wykonania i zaczyna je wykonywać od pierwszego do ostatniego wiersza.

Ostatnie zdanie nie jest jednak do końca prawdziwe, ponieważ interpreter najpierw przeszukuje plik w celu znalezienia instrukcji `BEGIN`, których kod wykonuje najpierw. Następnie wraca do pierwszego wiersza i zaczyna wykonywanie sekwencyjne. Więcej na temat instrukcji `BEGIN` znajduje się w podrozdziale 5.7.

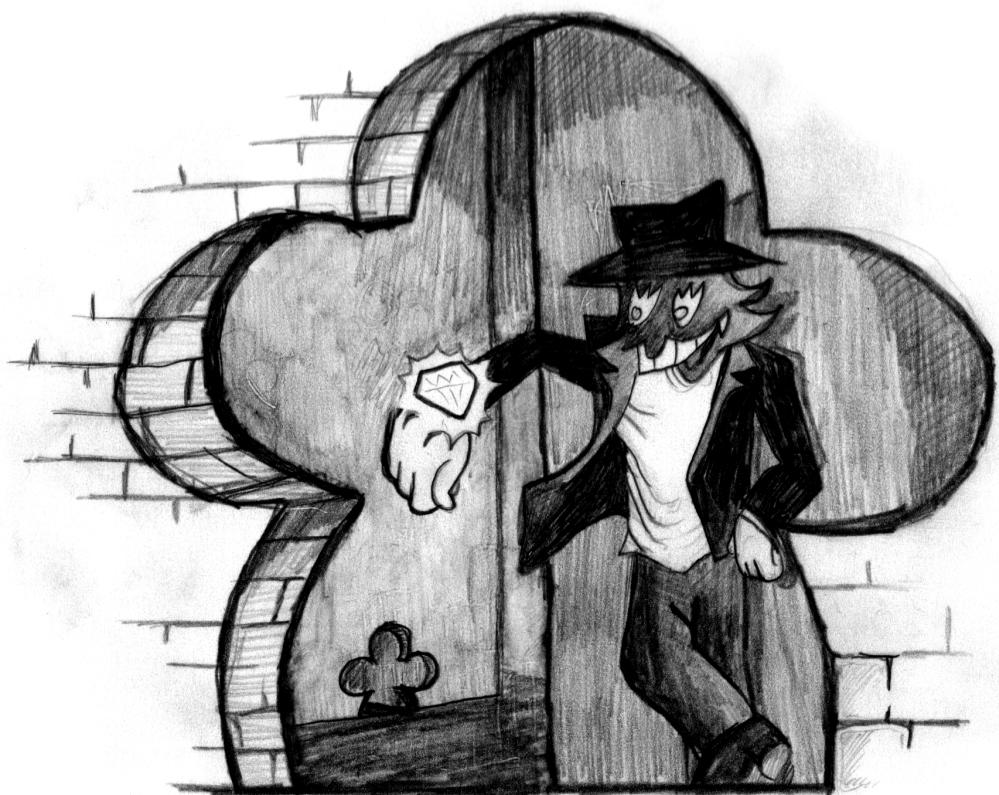
Inna różnica pomiędzy językiem Ruby a językami kompilowanymi dotyczy definicji modułów, klas i metod. W językach kompilowanych są to struktury syntaktyczne przetwarzane przez kompilator. W Ruby są to instrukcje jak wszystkie inne. Kiedy interpreter Ruby napotka definicję klasy, wykonuje ją, powodując powstanie nowej klasy. Podobnie kiedy napotka definicję metody, wykonuje ją, powodując powstanie nowej metody. W dalszej części programu interpreter najprawdopodobniej znajdzie wyrażenie wywołujące tę metodę, które spowoduje wykonanie instrukcji zawartych w ciele tej metody.

Interpreter Ruby wywoływany jest w wierszu poleceń poprzez podanie mu skryptu do wykonania. Bardzo proste jednowierszowe skrypty wpisuje się czasami bezpośrednio w wierszu poleceń. Częściej jednak podaje się nazwę pliku zawierającego skrypt. Interpreter Ruby odczytuje plik i wykonuje znajdujący się w nim skrypt. Najpierw wykonuje bloki `BEGIN`. Później przechodzi do pierwszego wiersza pliku i działa, dopóki nie wystąpi jedna z poniższych sytuacji:

- Interpreter wykona polecenie, które spowoduje zakończenie programu.
- Dojdzie do końca pliku.
- Dojdzie do wiersza oznaczającego logiczny koniec pliku za pomocą tokenu `_END_`.

Przed zakończeniem działania interpreter Ruby zazwyczaj (jeśli nie została wywołana metoda `exit!`) wykonuje ciała wszystkich znalezionych instrukcji `END` oraz pozostały kod zamknięcia zarejestrowany za pomocą funkcji `at_exit`.

Typy danych i obiekty

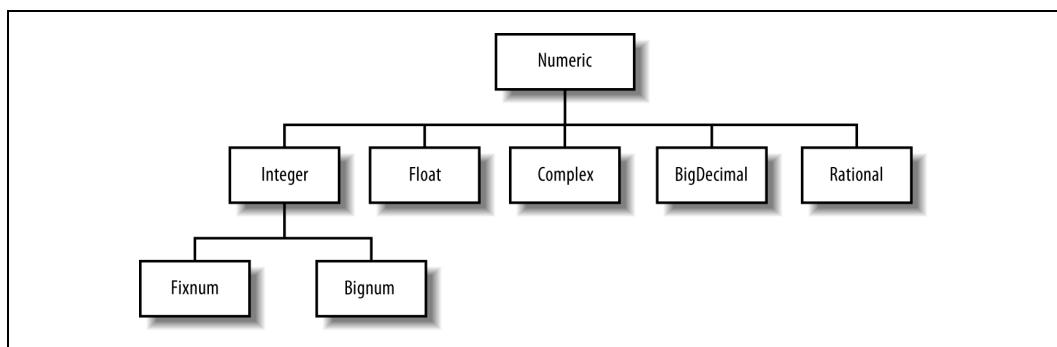


Aby opanować język programowania, konieczna jest znajomość dostępnych w nim struktur danych i operacji, które można na nich wykonywać. Niniejszy rozdział dotyczy wartości przetwarzanych przez programy Ruby. Na początku znajduje się pełny opis wartości liczbowych i tekstowych. Następne są tablice jednowymiarowe (tablice) i tablice asocjacyjne (haszowe) — dwie bardzo ważne struktury danych stanowiące podstawę języka Ruby. Dalej opisane są zakresy, symbole i specjalne wartości `true`, `false` oraz `nil`. W Ruby wszystkie wartości są obiektami, dlatego na końcu niniejszego rozdziału znajduje się szczegółowy opis własności wszystkich obiektów.

Klasy opisane w tym rozdziale są podstawowymi typami danych języka Ruby. Wyjaśnione zostały podstawy ich działania — sposób zapisu wartości literałowych w programie, arytmetyka liczb całkowitych i zmiennoprzecinkowych, kodowanie danych tekstowych, wartości jako klucze w tablicach asocjacyjnych itd. Mimo iż rozdział ten opisuje liczby,łańcuchy oraz tablice i tablice asocjacyjne, nie zawiera opisów ich API. Te wraz z przykładami znajdują się w rozdziale 9., w którym przedstawione są także inne ważne (ale nie podstawowe) klasy.

3.1. Liczby

W Ruby dostępnych jest pięć standardowych klas reprezentujących liczby, a biblioteka standardowa udostępnia jeszcze trzy dodatkowe, które również bywają przydatne. Rysunek 3.1 przedstawia hierarchię tych klas.



Rysunek 3.1. Hierarchia klas liczbowych

Wszystkie obiekty liczbowe w języku Ruby są egzemplarzami klasy `Numeric`. Wszystkie liczby całkowite są egzemplarzami klasy `Integer`. Jeśli liczba całkowita mieści się w 31 bitach (w większości implementacji), jest egzemplarzem klasy `Fixnum`. W przeciwnym przypadku jest typu `Bignum`. Obiekty tego typu mogą reprezentować liczby całkowite każdej wielkości. Dlatego jeżeli wynikiem działań wykonywanych na liczbach typu `Fixnum` jest liczba niemieszcząca się w tym zakresie, wynik jest konwertowany na typ `Bignum`. Podobnie jeśli wynik działań na typach `Bignum` mieści się w zakresie `Fixnum`, jest on zapisywany jako typ `Fixnum`. Liczby rzeczywiste są reprezentowane przez klasę `Float`, która wykorzystuje natywną reprezentację liczb zmiennoprzecinkowych platformy. Klasa `Complex` reprezentuje liczby złożone, `BigDecimal` — liczby rzeczywiste o dowolnej precyzji przy użyciu reprezentacji dziesiętnej zamiast binarnej, `Rational` — liczby wymierne (jedna liczba całkowita podzielona przez inną liczbę całkowitą). W Ruby 1.8 klasy te wchodzą w skład biblioteki standardowej. W Ruby 1.9 klasy `Complex` i `BigDecimal` są wbudowane.

Wszystkie obiekty liczbowe są **niemodyfikowalne** — nie istnieją metody pozwalające na zmianę wartości takiego obiektu. Przekazując referencję do obiektu liczbowego metodzie, nie trzeba obawiać się, że obiekt ten zostanie przez nią zmodyfikowany. Obiekty typu `Fixnum` są używane bardzo często. Dlatego różne implementacje Ruby traktują je jako bezpośrednie wartości, a nie jako referencje. Jednak przez to, że liczby są niemodyfikowalne, nie da się ich rozróżnić.

3.1.1. Literały całkowitoliczbowe

Literał całkowitoliczbowy nie jest niczym innym jak ciągiem cyfr.

```
0  
123  
12345678901234567890
```

Jeśli wartość całkowitoliczbową mieści się w zakresie klasy `Fixnum`, jest typu `Fixnum`. W przeciwnym przypadku jest typu `Bignum`, który może przechowywać liczby o dowolnej wielkości. Literały całkowitoliczbowe mogą zawierać znaki podkreślenia (nie na początku i końcu), które są czasami używane jako separatory części tysięcznych:

```
1_000_000_000 #Jeden miliard.
```

Jeżeli literał całkowitoliczbowy zaczyna się od zera i składa się z co najmniej dwóch cyfr, jest interpretowany jako liczba o podstawie innej niż 10. Liczby zaczynające się od znaków 0x lub 0X są szesnastkowe (o podstawie 16), a do reprezentacji cyfr od 10 do 15 wykorzystują litery a-f (lub A-F). Liczby zaczynające się od znaków 0b lub 0B są binarne (o podstawie 2) i składają się tylko z cyfr 0 i 1. Liczby zaczynające się od zera bez żadnych liter po nim są ósemkowe (o podstawie 8), a więc powinny składać się z cyfr od 0 do 7. Przykłady:

```
0377 #Ósemkowa reprezentacja liczby 255.  
0b1111_1111 #Binarna reprezentacja liczby 255.  
0xFF #Szesnastkowa reprezentacja liczby 255.
```

Aby utworzyć liczbę ujemną, wystarczy postawić przed nią znak minusa. Literały mogą także zaczynać się od plusa, ale to nic nie zmienia.

3.1.2. Literały liczb zmiennoprzecinkowych

Literał liczby zmiennoprzecinkowej składa się z nieobowiązkowego znaku plusa lub minusa, przynajmniej jednej cyfry dziesiętnej, kropki dziesiętnej (znaku .), co najmniej jednej cyfry za kropką i opcjonalnego wykładnika potęgi. Wykładnik zaczyna się od litery e lub E, po której znajduje się opcjonalny znak i co najmniej jedna cyfra dziesiętna. Podobnie jak w przypadku liczb całkowitych, można używać znaków podkreślenia jako separatorów. W przeciwieństwie do liczb całkowitych liczb zmiennoprzecinkowych nie można zapisywać w innym systemie niż dziesiętny. Poniżej znajduje się kilka przykładów literałów liczb zmiennoprzecinkowych:

```
0.0  
-3.14  
6.02e23 #Oznacza  $6.02 \times 10^{23}$ .  
1_000_000.01 #Troszkę więcej niż jeden milion.
```

W języku Ruby cyfry muszą znajdować się zarówno przed, jak i po kropce dziesiętnej. Nie można na przykład napisać .1. Jest to konieczne ze względu na uniknięcie dwuznaczności w skomplikowanej gramatyce tego języka. Ruby różni się pod tym względem od wielu innych języków programowania.

3.1.3. Arytmetyka

Wszystkie typy liczbowe definiują standardowe operatory `+`, `-`, `*` i `/` służące odpowiednio do dodawania, odejmowania, mnożenia i dzielenia. Jeśli wynik działania na liczbach całkowitych nie mieści się w zakresie Fixnum, zostaje automatycznie przekonwertowany na klasę Bignum. Dzięki temu działania arytmetyczne nigdy nie powodują przekroczenia zakresu, jak to ma miejsce w innych językach. Nadmiar arytmetyczny liczb zmiennoprzecinkowych (przynajmniej na platformach używających standardowej reprezentacji liczb zmiennoprzecinkowych IEEE-754) jest reprezentowany przez specjalną dodatnią lub ujemną nieskończoność, a niedomiar przez wartość zerową.

Działanie operatora dzielenia zależy od klasy jego operandów. Jeżeli oba są liczbami całkowitymi, wykonywane jest dzielenie z odrzuceniem części ułamkowej. W przypadku gdy którykolwiek z operandów jest typu zmiennoprzecinkowego, wykonywane jest dzielenie z zachowaniem części ułamkowej. Istnieją także trzy sposoby dzielenia: `div` wykonuje dzielenie całkowitoliczbowe, `fdiv` — dzielenie zmiennoprzecinkowe, a `quo`, kiedy to możliwe, zwraca obiekt `Rational`, w przeciwnym wypadku zwraca `Float` (wymaga to użycia modułu `Rational` w Ruby 1.8).

```
[5/2, 5.0/2, 5/2.0]          # Wynik = [2, 2.5, 2.5].
[5.0.div(2), 5.0.fdiv(2), 5.0.quo(2)]    # Wynik = [2, 2.5, Rational(5,2)].
```

Dzielenie liczby całkowitej przez zero powoduje błąd `ZeroDivisionError`. Dzielenie przez zero liczby zmiennoprzecinkowej nie powoduje błędu — zwracana jest wartość `Infinity`. Przypadek dzielenia `0.0/0.0` jest specjalny. W większości urządzeń i systemów operacyjnych jego wynikiem jest specjalna wartość zmiennoprzecinkowa o nazwie `NaN` (*Not-a-Number*).

Operator modulo `%` (a także analogiczna metoda `modulo`) oblicza resztę z dzielenia liczb całkowitych. Może być również używany z liczbami `Float` oraz `Rational`. Metoda `divmod` zwraca jednocześnie wynik dzielenia całkowitoliczbowego oraz resztę z dzielenia:

```
x = 5%2                      # Wynik = 1, wynik dzielenia to 2, reszta z dzielenia to 1.
q, r = 10.divmod 3            # Wynik = [3, 1], wynik dzielenia to 3, reszta z dzielenia to 1.
```

Dzielenie, modulo i liczby ujemne

Kiedy jeden operand (ale nie oba) jest ujemny, Ruby wykonuje dzielenie i operację modulo o innym charakterze niż języki C, C++ czy Java (ale takim samym jak Python czy Tcl). Weź pod uwagę iloraz $-7/3$. Wynik zmiennoprzecinkowy to -2.33 . Jednak wynikiem dzielenia liczb całkowitych musi być liczba całkowita, a więc trzeba go zaokrąglić. Ruby zaokrąglą w stronę ujemnej nieskończoności, a więc w tym przypadku do -3 . Język C i inne tego typu zaokrąglają w stronę zera, a więc zwrociłyby -2 (jest to tylko jeden ze sposobów charakteryzowania wyników — w rzeczywistości nie jest wykonywane żadne dzielenie zmiennoprzecinkowe).

Ważnym następstwem takiego sposobu dzielenia liczb całkowitych w języku Ruby jest fakt, że $-a/b = a/-b$, ale nie musi być równe $-(a/b)$.

Dzielenie modulo w języku Ruby również jest inne niż w takich językach jak C i Java. W Ruby $-7\%3$ daje wynik 2 . W C i Javie wynik ten będzie wynosił -1 . Rozmiar tej różnicy może być inny w zależności od wartości dzielonych liczb. Inny jest też znak wyniku. W języku Ruby znak wyniku jest zawsze taki sam jak znak drugiego operandu. W C i Javie znak wyniku odpowiada znakowi pierwszego operandu (w Ruby jest też dostępna metoda o nazwie `remainder`, która działa tak samo pod względem rozmiaru i znaku jak operator modulo w języku C).

Dodatkowo Ruby zapożyczył operator potęgowania `**` z języka Fortran. Wykładnik potęgi nie musi być liczbą całkowitą:

```
x**4          # To samo co x*x*x*x.  
x**-1         # To samo co 1/x.  
x**(1/3.0)    # Pierwiastek sześcienny z x.  
x**(1/4)      # Ups! Dzielenie liczb całkowitych oznacza, że jest to x**0, czyli zawsze 1.  
x**(.1/0.4)   # Pierwiastek czwartego stopnia z x.
```

Jeśli kilka działań potęgowania jest połączonych w jednym wyrażeniu, są one wykonywane od prawej do lewej. Zatem `4**3**2` oznacza to samo co `4**9`, a nie `64**2`.

Wynik potęgowania może być bardzo duży. Należy pamiętać, że liczby całkowite mogą mieć dowolną wielkość, ale obiekty typu `Float` nie mogą reprezentować liczb większych niż `Float::MAX`. Dlatego działanie `10**1000` da dokładny wynik całkowitoliczbowy, ale wyrażenie `9.9**1000` spowoduje nadmiar arytmetyczny i jego wartością będzie wartość `Infinity` typu `Float`.

Wartości klas `Fixnum` i `Bignum` obsługują standardowe operatory bitowe — `~, &, |, ^, >>` oraz `<<` dostępne w C, Javie i wielu innych językach programowania (szczególny na ten temat znajdują się w podrozdziale 4.6). Dodatkowo liczby całkowite można indeksować jak tablice w celu sprawdzenia (ale nie ustawienia) poszczególnych bitów. Indeks 0 zwraca najmniej znaczący bit:

```
even = (x[0] == 0) # Liczba jest parzysta, jeśli najmniej znaczący bit jest 0.
```

3.1.4. Liczby zmiennoprzecinkowe a błąd zaokrąglania

Większość zarówno sprzętu komputerowego, jak i języków programowania (także Ruby) zaokrąga liczby rzeczywiste przy użyciu reprezentacji zmiennoprzecinkowej, tak jak klasa `Float` w Ruby. Ze względu na wydajność sprzętową gros reprezentacji zmiennoprzecinkowych jest binarnych. Za ich pomocą można precyjnie przedstawić takie ułamki jak `1/2`, `1/4` czy `1/1024`. Niestety, ludzie najczęściej używają ułamków typu `1/10`, `1/100`, `1/1000` itd. (zwłaszcza w obliczeniach finansowych). Binarne liczby zmiennoprzecinkowe nie nadają się do precyjnego przedstawiania liczb takich jak `0,1`.

Obiekty typu `Float` są bardzo precyjne, dzięki czemu mogą przedstawić ułamek `0,1` w bardzo dużym przybliżeniu, ale brak możliwości dokładnej reprezentacji prowadzi do problemów. Spójrz na poniższe proste wyrażenie:

```
0.4 - 0.3 == 0.1 # W większości implementacji zwraca wartość false.
```

Z powodu błędu zaokrąglania różnica pomiędzy przybliżonymi wartościami `0,4` i `0,3` nie jest identyczna z przybliżoną reprezentacją ułamka `0,1`. Problem ten nie dotyczy tylko języka Ruby. Pojawia się on także we wszystkich językach reprezentujących liczby zmiennoprzecinkowe w standardzie IEEE-754, takich jak C, Java czy JavaScript.

Jednym z rozwiązań jest reprezentacja liczb rzeczywistych w systemie dziesiętnym, a nie binarnym. Przykładem takiej reprezentacji jest klasa `BigDecimal` z biblioteki standardowej Ruby. Jednak działania arytmetyczne na obiektach typu `BigDecimal` są wielokrotnie wolniejsze od działań na obiektach typu `Float`. Taka szybkość wystarczy dla standardowych obliczeń finansowych, ale jest zbyt mała do zastosowań naukowych. W podrozdziale 9.3.3 znajduje się krótka przykładowa użycia klasy `BigDecimal`.

3.2. Tekst

Tekst w języku Ruby jest reprezentowany przez obiekty klasy `String`. Można je modyfikować, a ponadto klasa ta udostępnia całą gamę metod służących do wydobywania podłańcuchów, wstawiania i usuwania tekstu, przeszukiwania, podmieniania itd. Język Ruby oferuje kilka sposobów przedstawiania literalów łańcuchowych w programach. Niektóre z nich pozwalają na interpolację łańcuchów umożliwiającą zamianę wartości dowolnego wyrażenia Ruby na literał łańcuchowy. W kolejnych podrozdziałach opisane są literaly łańcuchowe i znakowe oraz operatory łańcuchowe. Opis całego API łańcuchów znajduje się w podrozdziale 9.1.

Wzorce tekstowe są w języku Ruby reprezentowane jako obiekty typu `Regexp`. Język ten umożliwia wstawianie wyrażeń regularnych bezpośrednio do programu. Na przykład wyrażenie `/[a-z]\d+/?` reprezentuje jedną małą literę z co najmniej jedną cyfrą. Wyrażenia regularne są powszechnie używane w języku Ruby, ale nie stanowią podstawowego typu danych, jak liczby, łańcuchy czy tablice. Dokumentacja składni wyrażeń regularnych i API klasy `Regexp` znajduje się w podrozdziale 9.2.

Tekst w Ruby 1.8 i 1.9

Największa różnica pomiędzy Ruby 1.8 i 1.9 jest taka, że ostatnia wersja w pełni obsługuje Unicode i inne wielobajtowe systemy reprezentacji tekstu. Bardzo znamienne konsekwencje tego faktu zostały opisane w różnych miejscach niniejszego podrozdziału, a w szczególności w podrozdziale 3.2.6.

3.2.1. Literaly łańcuchowe

W języku Ruby istnieje kilka sposobów na wstawienie łańcucha do programu.

3.2.1.1. Literaly łańcuchowe w cudzysłowach pojedynczych

Najprostsze literaly łańcuchowe znajdują się w pojedynczych cudzysłowach (apostrofach). Tekst znajdujący się między tymi znakami stanowi wartość łańcucha:

'To jest prosty literał łańcuchowy.'

Aby w literale łańcuchowym otoczonym pojedynczymi cudzysłowami użyć apostrofu, należy postawić przed nim lewy ukośnik, który zapobiega potraktowaniu go przez interpreter jako koniec literalu:

'Tłumaczymy książki wydawnictwa O'Reilly.'

Lewy ukośnik służy także do wstawienia lewego ukośnika w łańcuchu. Jeśli są dwa ukośniki jeden po drugim, drugi nie jest traktowany jako symbol unikowy. Oto kilka sytuacji, w których konieczne jest użycie podwójnych lewych ukośników:

'Niniejszy literał łańcuchowy kończy się jednym ukośnikiem: \\'

'Ten kończy się lewym ukośnikiem i apostrofem: \\\\'

'Dwa lewe ukośniki: \\\\\'

W łańcuchach otoczonych pojedynczymi cudzysłowami lewy ukośnik nie jest znakiem specjalnym, jeżeli nie znajduje się za nim kolejny lewy ukośnik lub pojedynczy cudzysłów. Z tego względu w większości przypadków lewe ukośniki w literałach łańcuchowych nie muszą

występować w parach (aczkolwiek mogą). Na przykład poniższe dwa literały łańcuchowe są równoważne:

```
'a\b' == 'a\\b'
```

Łańcuchy w pojedynczych cudzysłowach mogą zajmować kilka linijek, a powstały literał łańcuchowy zawiera znaki nowego wiersza. Nie ma możliwości uniknięcia wstawienia tych znaków za pomocą lewego ukośnika:

```
'To jest długi literał łańcuchowy \,  
który zawiera lewy ukośnik i znak nowego wiersza.'
```

Aby podzielić długi literał łańcuchowy otoczony pojedynczymi cudzysłowami na kilka linii bez wstawiania znaków nowego wiersza, należy rozbić go na kilka literałów. Interpreter połączyc je podczas analizy składniowej. Należy jednak pamiętać o zajęciu się znakami nowego wiersza (zobacz rozdział 2.) pomiędzy literałami, które interpreter może potraktować jako zaakończenie instrukcji:

```
message =  
'Te trzy literały są '\  
'połączone przez interpreter w jeden łańcuch. '\  
'Powstały łańcuch nie zawiera żadnych znaków nowego wiersza.'
```

3.2.1.2. Literały łańcuchowe w cudzysłowach podwójnych

Literały łańcuchowe w podwójnych cudzysłowach są znacznie bardziej elastyczne od tych otoczonych cudzysłowami pojedynczymi. Pozwalają na używanie kilku znaków specjalnych, takich jak `\n` — nowy wiersz, `\t` — tabulator, `\\"` — cudzysłów nieoznaczający końca łańcucha:

```
"\t\" Ten cytat zaczyna się od tabulatora, a kończy znakiem nowego wiersza.\\"n"  
"\\" #Jeden lewy ukośnik.
```

W Ruby 1.9 znak specjalny `\u` wstawia do łańcucha w podwójnych cudzysłowach dowolny znak Unicode określony za pomocą jego punktu kodowego. Jest to na tyle skomplikowane zagadnienie, że poświęcony mu został osobny podrozdział (3.2.1.3.). Inne znaki specjalne wykorzystujące lewy ukośnik są rzadko używane, a służą do kodowania danych binarnych w łańcuchach. Pełną listę znaków specjalnych przedstawia tabela 3.1.

Co więcej, literały łańcuchowe otoczone podwójnymi cudzysłowami mogą zawierać dowolne wyrażenia Ruby. Przy tworzeniu łańcucha obliczana jest wartość takiego wyrażenia, następuje jej konwersja na łańcuch i zostaje ona wstawiona do niego w miejscu wyrażenia. Takie zastępowanie wyrażeń ich wartościami nazywane jest interpolacją łańcuchów. Wyrażenia w łańcuchach otoczonych podwójnymi cudzysłowami zaczynają się od znaku `#` i są otoczone nawiasami klamrowymi:

```
"360 stopni=#{2*Math::PI} radianów" # "360 stopni=6.28318530717959 radianów".
```

Jeśli wyrażenie, które ma zostać interpolowane w literale łańcuchowym, jest referencją do zmiennej globalnej, obiektowej lub klasowej, nawiasy klamrowe można opuścić:

```
$salutation = 'Witaj'          # Definicja zmiennej globalnej.  
"#$salutation kolego"        # Użycie zmiennej globalnej w łańcuchu otoczonym podwójnymi cudzysłowami.
```

Aby uniknąć specjalnego traktowania znaku `#`, należy postawić przed nim lewy ukośnik `\`. Warto pamiętać, że jest to konieczne tylko wówczas, gdy po znaku `#` znajduje się `{`, `$` lub `@`:

```
"Mój numer telefonu #: 555-1234"      # Brak znaku specjalnego.  
"Do interpolacji wyrażeń używaj \#{ " # Łańcuch #{ ukryty za pomocą lewego ukośnika.
```

Interpolacja łańcuchów za pomocą funkcji sprintf

Programiści języka C mogą ucieszyć się na wieść, że także w języku Ruby dostępne są funkcje `printf1` i `sprintf` służące do interpolacji sformatowanych wartości do łańcuchów:
`sprintf("pi wynosi około %.4f", Math::PI) # Zwraca "pi wynosi około 3.1416".`

Zaletą tego rodzaju interpolacji jest to, że łańcuch formatujący może określać opcje takie jak liczba miejsc po przecinku w obiektach typu `Float`. W prawdziwym stylu Ruby istnieje nawet forma operatorowa metody `sprintf` — należy pomiędzy łańcuchem formatującym a argumentami, które mają być w nim interpolowane, wstawić znak %:

```
"pi wynosi około %.4f" % Math::PI # To samo co powyżej.  
"%s: %f" % ["pi", Math::PI] # Tablica po prawej stronie dla wielu argumentów.
```

Literał łańcuchowe otoczone podwójnymi cudzysłowniami mogą zajmować kilka linii; znaki końca wiersza wchodzą w skład literału, chyba że zostaną stłumione za pomocą lewego ukośnika:

*"Ten literał łańcuchowy
zajmuje dwa wiersze,\n mimo że został zapisany w trzech linijkach."*

Czasami potrzebne jest jawnie wpisanie znaków końca wiersza w łańcuchu — na przykład w celu wymuszenia znaków końca wiersza `CRLF` (*Carriage Return Line Feed*), które są stosowane w protokole HTTP. Aby to zrobić, należy wszystkie literaly łańcuchowe zapisywać w jednej linii i wstawiać na końcu każdej z nich znaki specjalne `\r` i `\n`. Nie należy zapominać, że bezpośrednio sąsiadujące literaly łańcuchowe są łączone, ale jeśli zostaną zapisane w osobnych liniach, konieczne jest stłumienie znaku nowego wiersza pomiędzy nimi:

*"Ten łańcuch ma trzy wiersze.\r\n" \n
"Jest zapisany jako trzy sąsiadujące ze sobą literały\r\n" \n
"oddzielone stłumionymi znakami nowego wiersza.\r\n"*

3.2.1.3. Wstawianie znaków Unicode

W Ruby 1.9 istnieje możliwość wstawiania do łańcuchów otoczonych podwójnymi cudzysłowniami dowolnych znaków Unicode przy użyciu sekwencji `\u`. W najprostszej formie po łańcuchu `\u` występują cztery cyfry w zapisie szesnastkowym (litery mogą być małe lub wielkie) reprezentujące punkt kodowy Unicode z zakresu 0000 – FFFF. Na przykład:

```
"\u00D7"      # => "×": wiodących zer nie można opuścić.  
"\u20ac"      # => "€": małe litery są również dozwolone.
```

Inny format sekwencji specjalnej `\u` składa się z nawiasów klamrowych, w których znajduje się od jednej do sześciu cyfr w zapisie szesnastkowym. Cyfry w klamrach mogą reprezentować dowolny punkt kodowy z zakresu 0-10FFFF, a wiodące zera można opuścić:

```
"\u{A5}"       # => "¥": to samo co "u00A5".  
"\u{3C0}"      # Mała grecka litera pi: to samo co "u03C0".  
"\u{10ffff}"   # Największy punkt kodowy Unicode.
```

Na koniec należy dodać, że format `\u{}` pozwala na wstawienie kilku punktów kodowych za jednym razem. W tym celu należy w klamrach wpisać dowolną liczbę zestawów sześciu cyfr w zapisie szesnastkowym oddzielonych pojedynczą spacją lub pojedynczym tabulatorem. Po otwierającej klamrze i przed klamrą zamkającą nie może być spacji:

```
money = "\u{20AC} A3 A5" # => "€A3 A5"
```

¹ Więcej na jej temat można się dowiedzieć, korzystając z narzędzia `ri Kernel.printf`.

Tabela 3.1. Znaki specjalne stosowane w łańcuchach otoczonych podwójnymi cudzysłowami

Znak specjalny	Znaczenie
\x	Lewy ukośnik znajdujący się przed dowolnym znakiem x odpowiada temu znakowi x, chyba że x jest znakiem końca wiersza lub jednym ze znaków specjalnych abcfnrstuvwxyzCM01234567. Składnia ta pozwala uniknąć specjalnego znaczenia znaków \, # i ".
\a	Znak BEL (kod ASCII 7). Odtwarza dźwięk konsoli. Jest równoważny z \C-g lub \007.
\b	Znak Backspace (kod ASCII 8). Równoważny z \C-h lub \010.
\e	Znak ESC (kod ASCII 27). Równoważny z \C-1 i \014.
\f	Znak wysuwu strony (kod ASCII 12). Równoważny z \C-1 i \014.
\n	Znak nowego wiersza (kod ASCII 10). Równoważny z \C-j i \012.
\r	Znak powrotu karetki (kod ASCII 13). Równoważny z \C-m lub \015.
\s	Znak spacji (kod ASCII 32).
\t	Znak tabulatora (kod ASCII 9). Równoważny z \C-i i \011.
\unnnn	Punkt kodowy Unicode, gdzie każde n jest cyfrą szesnastkową. Wiodące zera nie mogą zostać opuszczone. Wszystkie cztery cyfry są wymagane. Obsługiwany w Ruby 1.9 i dalszych.
\u{hexdigits}	Punkt kodowy (lub punkty kodowe) Unicode określony przez cyfry szesnastkowe hexdigits. Opis tego znaku specjalnego znajduje się w głównym tekście. Obsługiwany w Ruby 1.9 i dalszych.
\v	Znak tabulatora pionowego.
\nnnn	Bajt nnn, gdzie nnn to trzy cyfry w zapisie ósemkowym z zakresu 000-377.
\nnn	To samo co Onn, gdzie nn to dwie cyfry w zapisie ósemkowym z zakresu 00-77.
\n	To samo co 0On, gdzie n jest cyfrą w zapisie ósemkowym z zakresu 0-7.
\xnn	Bajt nn, gdzie nn to dwie cyfry w zapisie szesnastkowym z zakresu 00-FF (cyfry szesnastkowe mogą być zapisywane zarówno małymi, jak i wielkimi literami).
\xn	To samo co \x0n, gdzie n jest cyfrą szesnastkową z przedziału 0-F (lub f).
\cx	Skrót od \C-x.
\C-x	Znak, którego kod jest utworzony poprzez wyzerowanie szóstego i siódmego bitu x przy zachowaniu wysokiego bitu i pięciu bitów niskich. x może być dowolnym znakiem, ale sekwencja ta jest zazwyczaj używana do reprezentacji znaków sterujących od A do Z (kody ASCII od 1 do 26). Ze względu na układ tabeli ASCII można używać zarówno małych, jak i wielkich liter w miejscu x. Należy zauważyć, że skrócony zapis to \cx. x może być dowolnym pojedynczym znakiem lub znakiem specjalnym innym niż \C\u, \x i \nnn. \M można połączyć z \C.
\M-x	Znak, którego kod jest utworzony poprzez ustawienie wysokiego bitu kodu x. Używany do reprezentacji metaznaków z technicznego punktu widzenia niewchodzących w skład zestawu ASCII. x może być dowolnym pojedynczym znakiem lub znakiem specjalnym innym niż \M\u, \x i \nnn. \M można połączyć z \C.
\eol	Lewy ukośnik przed znakiem końca wiersza tłumia działanie tego znaku. Ani ukośnik, ani znak końca wiersza nie pojawiają się w łańcuchu.

Należy zauważyć, że spacje znajdujące się w obrębie klamer nie kodują spacji w samym łańcuchu. Można jednak zakodować spację ASCII, używając punktu kodowego Unicode 20:

```
money = "\u{20AC} 20 A3 20 A5" # => "€ £ ¥"
```

Łańcuchy zawierające sekwencje specjalne \u są zakodowane w systemie UTF-8 (więcej informacji na temat kodowania łańcuchów znajduje się w podrozdziale 3.2.6).

Sekwencje specjalne \u zazwyczaj, ale nie zawsze, można stosować w łańcuchach. Jeśli plik jest zakodowany w innym systemie niż UTF-8, a łańcuch zawiera znaki wielobajtowe w tym kodowaniu (dosłownie znaki — nieutworzone za pomocą znaków specjalnych), to w takim

łańcuchu nie można używać \u — jeden łańcuch nie może kodować znaków w dwóch różnych systemach. Sekwencji specjalnej \u można używać we wszystkich dokumentach z kodowaniem źródła (zobacz podrozdział 2.4.1) UTF-8. Można ją stosować zawsze w łańcuchach znaków ASCII.

Sekwencji specjalnej \u można używać także w innego typu tekstach (o których za chwilę), takich jak wyrażenia regularne, literały znakowe, łańcuchy otoczone ogranicznikami % - i %Q, tablice ograniczone znakami %W, dokumenty miejscowe oraz łańcuchy poleceń otoczone odwrotnymi apostrofami. Programiści Javy powinni zauważyc, że sekwencja specjalna \u w języku Ruby może znajdować się wyłącznie w tekście umieszczonym w jakieś rodzaju cudzysłowach, nie w identyfikatorach programu.

3.2.1.4. Dowolne ograniczniki literałów łańcuchowych

Jeżeli tekst zawiera apostrofy i cudzysłowy, używanie go jako literalu w podwójnych lub pojedynczych cudzysłowach nie jest eleganckim rozwiązaniem. Język Ruby udostępnia ogólną składnię do cytowania literałów łańcuchowych (oraz, jak przekonasz się dalej, także do literałów wyrażeń regularnych i tablicowych). Sekwencja %q zaczyna literał łańcuchowy, który podlega zasadom łańcuchów w cudzysłowach pojedynczych. Natomiast %Q (lub tylko %) otwiera literały podlegające zasadom łańcuchów w cudzysłowach podwójnych. Pierwszy znak znajdujący się za znakiem q lub Q określa znak ograniczający. Literał ciągnie się do napotkania pasującego (niewstawionego za pomocą znaku specjalnego) ogranicznika. Gdy ogranicznikiem otwierającym jest (, [, { lub <, wtedy jego odpowiednikiem jest),], } lub > (należy zauważyc, że odwrotny apostrof ` nie jest parą dla apostrofu '). W przeciwnym przypadku ogranicznik zamykający jest taki sam jak otwierający. Oto kilka przykładów:

```
%q(Nie trzeba przejmować się zastępowaniem znaków '!)  
%Q|"Co słychać", spytał|  
%-Ten literał łańcuchowy kończy się znakiem nowego wiersza\n- #Pominięto Q.
```

Jeśli zajdzie konieczność zastosowania znaku ograniczającego w tekście, można posłużyć się lewym ukośnikiem (nawet w surowszej wersji %q) lub użyć innego ogranicznika:

```
%q_Ten literał łańcuchowy zawiera \_znaki podkreślenia\_\_\_  
%Q!Wystarczy użyć \_innego ogranicznika\!!
```

Znaków będących ogranicznikami można używać bez stosowania lewego ukośnika w środku literalu, pod warunkiem że są używane w odpowiadających sobie poprawnie zagnieżdżonych parach:

```
# W XML używane są pary nawiasów kątowych:  
%<<book><title>Ruby in a Nutshell</title></book>> # To działa.  
# Wyrażenia znajdują się w nawiasach:  
%((1+(2*3)) = #{(1+(2*3))}) # To także działa.  
%(Nawias bez pary \ musi być zastąpiony sekwencją specjalną) # Tutaj potrzebny jest lewy ukośnik.
```

3.2.1.5. Dokumenty miejscowe

W przypadku długich literałów łańcuchowych może nie udać się znaleźć znaku ograniczającego, którego w którymś miejscu nie będzie trzeba zastępować sekwencją specjalną. Rozwiązanie tego problemu w języku Ruby polega na określaniu dowolnej sekwencji znaków, które będą służyć jako ogranicznik. Ten rodzaj literalu został zapożyczony ze składni powłoki systemu Unix i jest znany pod nazwą dokumentu lokalnego (ang. *here document*) — ponieważ dokument ten znajduje się bezpośrednio w pliku z kodem źródłowym, a nie w osobnym pliku.

W poniższym przykładzie dokumenty zaczynają się od znaków << lub <<- . Bezpośrednio po nich (nie może być ani jednej spacji, aby uniknąć pomyłki z operatorem przesunięcia w lewo) znajduje się identyfikator lub łańcuch określający ogranicznik końcowy. Tekst literału zaczyna się w następnym wierszu i ciągnie się aż do napotkania ogranicznika końcowego. Na przykład:

```
document = <<HERE          # Początek dokumentu miejscowego.  
Literał łańcuchowy.  
Składa się z dwóch linijek, które nagle się kończą...  
HERE
```

Interpretaator pobiera treść literału łańcuchowego po jednym wierszu. Nie oznacza to jednak, że znaki << muszą być ostatnie w swojej linijce. W rzeczywistości interpreter po wczytaniu treści dokumentu lokalnego wraca do wiersza, w którym był, i kontynuuje jego przetwarzanie. Poniższy przykładowy kod Ruby tworzy łańcuch, łącząc dwa dokumenty lokalne i zwykły łańcuch otoczony pojedynczymi cudzysłowami:

```
greeting = <<HERE + <<THERE + "drogi"  
Witaj  
HERE  
kolego  
THERE
```

Ciąg znaków <<HERE w pierwszym wierszu powoduje, że interpreter wczytuje wiersze 2. i 3. Ciąg <<THERE powoduje wczytanie wierszy 4. i 5. Po wczytaniu tych wierszy wszystkie trzy literały zostają połączone w jeden łańcuch.

Końcowy ogranicznik dokumentu lokalnego musi być jedyny w linijce — nie może znajdować się po nim żaden komentarz. Jeśli dokument lokalny zaczyna się od znaków <<, ogranicznik musi znajdować się na początku linijki. W sytuacji gdy dokument zaczyna się od znaków <<-, przed ogranicznikiem mogą znajdować się białe znaki. Znak nowego wiersza na początku dokumentu lokalnego nie należy do literału, ale taki sam znak na końcu już tak. W związku z tym każdy dokument lokalny kończy się znakiem końca wiersza, z wyjątkiem pustych dokumentów lokalnych, które są równoważne z "":

```
empty = <<END  
END
```

Jeżeli jako znak końcowy zostanie użyty identyfikator bez cudzysłowów jak w poprzednich przykładach, dokument lokalny traktuje znaki lewego ukośnika i # jak łańcuch otoczony podwójnymi cudzysłowniami. Aby utworzyć identyfikator, który wykluczy potrzebę stosowania jakichkolwiek znaków specjalnych, a przy okazji umożliwić stosowanie spacji w ograniczniku, należy ogranicznik umieścić w pojedynczych cudzysłownikach:

```
document = <<'TO JEST KONIEC DROGI PRZYJACIELU, TO JEST KONIEC'  
. . .  
. . Dużo tekstu  
. . bez żadnych sekwencji specjalnych.
```

```
TO JEST KONIEC DROGI PRZYJACIELU, TO JEST KONIEC
```

Pojedyncze cudzysłowy otaczające ogranicznik wskazują, że ten literał łańcuchowy należy traktować jako łańcuch w pojedynczych cudzysłownikach. W rzeczywistości tego rodzaju dokument lokalny jest jeszcze bardziej ograniczony. Ponieważ pojedynczy cudzysłów nie jest ogranicznikiem, nigdy nie ma potrzeby zastępowania go znakiem specjalnym. Dodatkowo dzięki temu że lewy ukośnik nie jest nigdy potrzebny w charakterze znaku specjalnego, nigdy nie ma potrzeby zastępowania go specjalną sekwencją. W związku z tym w tego typu dokumentach lokalnych lewe ukośniki są zwykłymi znakami.

Ogranicznik dokumentu miejscowego można umieścić także w cudzysłowach podwójnych. Jedyna różnica w stosunku do zwykłego identyfikatora polega na możliwości stosowania spacji w ograniczniku:

```
document = <<- "# # "      # To jest jedynie miejsce, gdzie można umieścić komentarz.  
<html><head><title>#${title}</title></head>  
<body>  
<h1>#${title}</h1>  
#${body}  
</body>  
</html>  
# # #
```

Należy zauważyc, że w dokumentach miejscowych nie może być komentarzy z wyjątkiem pierwszej linijki, za tokenem <<, a przed początkiem literalu. Z wszystkich znaków # w powyższym fragmencie kodu jeden wstawia komentarz, trzy interpolują wyrażenia na literał, a pozostałe są znakami składającymi się na ogranicznik.

3.2.1.6. Wykonywanie poleceń w odwrotnych apostrofach

W języku Ruby istnieje jeszcze inną konstrukcję syntaktyczną związana z cudzysłowami i łańcuchami. Tekst znajdujący się pomiędzy odwrotnymi apostrofami (znak `) jest traktowany jako literał łańcuchowy otoczony cudzysłowami podwójnymi. Wartość takiego literala jest przekazywana do specjalnej metody Kernel.`. Wykonuje ona tekst jako polecenie powłoki systemu operacyjnego i zwraca wartość tego polecenia w postaci łańcucha.

Spójrz na następujący fragment kodu Ruby:

```
`ls`
```

W systemie Unix te cztery znaki tworzą łańcuch zwracający listę nazw plików znajdujących się w aktualnym katalogu. Oczywiście polecenia te są ściśle związane z konkretną platformą. Polecenie o podobnym działaniu w systemie Windows to `dir`.

Zamiast odwrotnych apostrofów można użyć uogólnionej składni cytowania. Przypomina ona opisaną wcześniej składnię %Q, ale używa znaków %x:

```
%x[ls]
```

Należy pamiętać, że tekst znajdujący się pomiędzy odwrotnymi cudzysłowami (lub za ciągiem %x) jest przetwarzany tak samo jak literał w podwójnych cudzysłowach. Oznacza to, że do łańcucha można interpolować dowolne wyrażenie Ruby. Na przykład:

```
if windows  
  listcmd = 'dir'  
else  
  listcmd = 'ls'  
end  
listing = `#{listcmd}`
```

W przypadku takim jak powyższy łatwiej jest bezpośrednio wywołać metodę Kernel.`:

```
listing = Kernel.(listcmd)
```

3.2.1.7. Modyfikowanie literałów łańcuchowych

W języku Ruby łańcuchy można modyfikować. Z tego względu interpreter Ruby nie może używać jednego obiektu do reprezentacji dwóch identycznych literałów łańcuchowych (jest to zaskakująca cecha dla programistów Javy). Kiedy Ruby natrafia na literał łańcuchowy,

zawsze tworzy nowy obiekt. Jeśli literał zostanie umieszczony w ciele pętli, Ruby utworzy nowy obiekt dla każdej iteracji. Można to sprawdzić za pomocą poniższej procedury:

```
10.times { puts "test".object_id }
```

Ze względów wydajnościowych należy unikać używania literałów w pętlach.

3.2.1.8. Metoda String.new

Poza wszystkimi wymienionymi dotychczas opcjami literałów łańcuchowych nowy łańcuch można utworzyć, korzystając z metody `String.new`. Jeśli nie zostaną podane żadne argumenty, metoda ta tworzy i zwraca nowy obiekt typu `String` niezawierający żadnych znaków. Jeżeli jako argument zostanie podany łańcuch, `String.new` zwraca obiekt typu `String` reprezentujący tekst podany jako argument.

3.2.2. Literaty znakowe

Pojedyncze znaki można wstawać do programów Ruby, poprzedzając je znakiem zapytania. Nie używa się cudzysłowów.

```
?A      # Literal znakowy znaku ASCII A.  
?"      # Literal znakowy znaku podwójnego cudzysłowu.  
??      # Literal znakowy znaku zapytania.
```

Mimo że język Ruby posiada specjalną konstrukcję składniową dla literałów łańcuchowych, brak w nim specjalnej klasy reprezentującej te literaty. Dodatkowo sposób ich interpretacji jest inny w wersji Ruby 1.9 niż w 1.8. W Ruby 1.8 wartością literala znakowego jest kod w postaci liczby całkowitej danego znaku. Na przykład literałowi `?A` odpowiada kod 65, ponieważ kod ASCII wielkiej litery A to 65. W Ruby 1.8 składnia literałów łańcuchowych działa tylko ze znakami ASCII i jednobajtowymi.

W Ruby 1.9 i późniejszych znaki są łańcuchami o długości 1. To znaczy że literał `?A` jest tym samym co literał 'A' i nie ma żadnej rzeczywistej potrzeby na stosowanie tej składni w nowym kodzie. W Ruby 1.9 składnia literałów znakowych działa ze znakami wielobajtowymi i może być używana z sekwencją specjalną Unicode `\u` (aczkolwiek nie z formą pozwalającą na użycie wielu punktów kodowych za jednym razem `\u{a b c}`):

```
?\u20AC == ?€      # => true: Tylko Ruby 1.9.  
?€ == "\u20AC"     # => true.
```

Składni literałów znakowych można używać w połączeniu z dowolnym ze znaków specjalnych wymienionych w tabeli 3.1:

```
?t      # Literal łańcuchowy znaku tabulatora.  
?\C-x    # Literal znakowy Ctrl-X.  
?\111    # Literal znaku, którego kod to 0111 (ósemkowy).
```

3.2.3. Operatory łańcuchowe

Klasa `String` udostępnia kilka przydatnych operatorów służących do wykonywania działań na łańcuchach tekstowych. Operator `+` łączy dwa łańcuchy i zwraca wynik w postaci nowego obiektu typu `String`:

```
planet = "Ziemia"  
"Witajcie na planecie" + " " + planet      # Zwraca łańcuch "Witajcie na planecie Ziemia."
```

Programiści Javy powinni zauważyc, że operator + nie konwertuje prawego operandu na łańcuch. Trzeba to zrobić własnoręcznie:

```
"Witajcie na planecie #" + planet_number.to_s #Metoda to_s konwertuje na łańcuch.
```

Oczywiście w języku Ruby zazwyczaj prostsza jest interpolacja łańcuchów niż ich konkatenacja za pomocą operatora +. W przypadku interpolacji metoda to_s jest wykonywana automatycznie:

```
"Witajcie na planecie ##{planet_number}"
```

Operator << dołącza swój drugi operand do pierwszego; powinien on być znany programistom C++. Różni się on znacznie od operatora +, ponieważ zmienia lewy operand, zamiast tworzyć i zwracać nowy obiekt:

```
greeting = "Witaj"  
greeting << " " << "świecie"  
puts greeting #Drukuje "Witaj świecie".
```

Operator <<, podobnie jak +, nie konwertuje typu prawego operandu. Jeśli jednak operand ten jest liczbą całkowitą, uznawany jest za kod znaku i dołączany jest odpowiadający temu kodowi znak. W Ruby 1.8 dozwolone są tylko liczby całkowite z przedziału 0-255. W Ruby 1.9 może zostać użyta każda liczba całkowita reprezentująca prawidłowy punkt kodowy:

```
alphabet = "A"  
alphabet << ?B # Teraz alfabet zawiera litery "AB".  
alphabet << 67 # A teraz "ABC".  
alphabet << 256 # Błąd w Ruby 1.8: kody muszą należeć do przedziału od 0 (włącznie) do 255.
```

Operator * przyjmuje po prawej stronie liczby całkowite. Zwraca obiekt typu String zawierający tyle wystąpień tekstu znajdującego się po jego lewej stronie, ile wskazuje liczba po prawej stronie:

```
ellipsis = '.*'3 # Zwraca wartość '...'
```

Jeśli po lewej stronie znajduje się literal łańcuchowy, wszelkie interpolacje są wykonywane przed każdym z powtórzeń. Oznacza to, że poniższy przekombinowany fragment kodu nie robi tego, czego można by było się po nim spodziewać:

```
a = 0;  
"#{a=a+1} " * 3 # Zwraca "1 1 1", a nie "1 2 3".
```

Klasa String udostępnia wszystkie standardowe operatory porównywania. Operatory == i != sprawdzają, czy dwa łańcuchy są równe i różne. Dwa łańcuchy są równe wtedy i tylko wtedy, gdy mają taką samą długość i takie same wszystkie znaki. Operatory <, <=, > i >= sprawdzają porządek względny łańcuchów, porównując kody składających się na nie znaków. Jeśli jeden łańcuch jest prefiksem drugiego, krótszy z nich jest mniejszy niż dłuższy. Porównywanie jest ściśle oparte na kodach znaków. Nie jest wykonywana żadna normalizacja, a kolejność w języku naturalnym (jeśli różni się od kolejności liczbowej kodów znaków) jest ignorowana.

Przy porównywaniu łańcuchów znaków rozróżniane są małe i wielkie litery². Należy pamiętać, że w ASCII wielkie litery mają niższe wartości kodów niż małe litery. Oznacza to na przykład, że "Z" < "a". Aby porównać znaki ASCII bez rozróżniania małych i wielkich liter, należy przed wykonaniem tej operacji wywołać na ich rzecz metodę downcase lub upcase (pamiętaj, że język Ruby rozpoznaje małe i wielkie litery tylko w zestawie ASCII).

² Ustawienie wycofywanej zmiennej globalnej \$= na wartość true w języku Ruby sprawia, że operatory porównania ==, < i inne tego typu nie rozróżniają małych i wielkich liter przy porównywaniu. Nie należy jednak tego robić. Takie ustawienie zmiennej powoduje zgłaszanie przez interpreter ostrzeżenia, nawet jeśli zostanie wywołany bez opcji -w. W Ruby 1.9 zmienna \$= nie jest już obsługiwana.

3.2.4. Dostęp do znaków i podłańcuchów

Najważniejszym z wszystkich operatorów obsługiwanych przez klasę `String` jest prawdopodobnie kwadratowy nawias; stosując go, można wydobywać lub zmieniać fragmenty łańcucha. Jest to dość elastyczny operator, którego można używać z operandami kilku różnych typów. Może też znajdować się po lewej stronie przypisania jako sposób na zmianę treści łańcucha.

W Ruby 1.8 łańcuch jest tablicą bajtów lub ośmiorabajtowych kodów znakowych. Rozmiar tej tablicy można sprawdzić za pomocą metody `length` lub `size`. Aby pobrać lub ustawić element tablicy, wystarczy wpisać odpowiedni znak w nawiasach kwadratowych:

```
s = 'hello';      # Ruby 1.8.  
s[0]           # 104: kod ASCII pierwszej litery 'h'.  
s[s.length-1]  # 111: kod ostatniej litery 'o'.  
s[-1]          # 111: inny sposób dostępu do ostatniego znaku.  
s[-2]          # 108: przedostatni znak.  
s[-s.length]   # 104: inny sposób dostępu do pierwszego znaku.  
s[s.length]    # nil: pod tym indeksem nie ma żadnego znaku.
```

Należy zauważać, że ujemne indeksy tablicy określają dostęp do elementów od końca. Warte uwagi jest również to, że Ruby nie zgłasza wyjątku przy próbie dostępu do znaku poza łańcuchem. W takiej sytuacji zwraca wartość `nil`.

Ruby 1.9 zwraca łańcuch składający się z jednego znaku zamiast kodu znaku. Warto pamiętać, że w przypadku wielorabajtowych łańcuchów zawierających znaki zakodowane przy użyciu różnej liczby bajtów swobodny dostęp do tych znaków jest znacznie wolniejszy niż do ich bajtów:

```
s = 'hello';      # Ruby 1.9.  
s[0]           # 'h': pierwszy znak łańcucha jako łańcuch.  
s[s.length-1]  # 'o': ostatni znak 'o'.  
s[-1]          # 'o': inny sposób dostępu do ostatniego znaku.  
s[-2]          # 'l': przedostatni znak.  
s[-s.length]   # 'h': inny sposób na dostęp do pierwszego znaku.  
s[s.length]    # nil: pod tym indeksem nie ma żadnego znaku.
```

Aby zmienić wybrany znak w łańcuchu, wystarczy użyć nawiasów kwadratowych po lewej stronie wyrażenia przypisania. W Ruby 1.8 po prawej stronie może znaleźć się znak ASCII lub łańcuch. W Ruby 1.9 po prawej stronie musi być łańcuch. Literałów znakowych można używać w obu wersjach języka:

```
s[0] = ?H        # Zamiana pierwszego znaku na H.  
s[-1] = ?O       # Zamiana ostatniego znaku na O.  
s[s.length] = ?? # Błąd! Nie można wykonać operacji przypisania poza łańcuchem.
```

Po prawej stronie takiej instrukcji przypisania nie może znaleźć się kod znaku — może to być dowolny łańcuch, wliczając łańcuchy wieloznakowe i puste. Dotyczy to zarówno Ruby 1.8, jak i 1.9:

```
s = "hello"      # Witasz się.  
s[-1] = ""        # Usuwasz ostatni znak. łańcuch s to teraz "hell".  
s[-1] = "p!"       # Zmieniasz ostatni znak i jeden dodajesz. łańcuch s to teraz "help!".
```

Znacznie częściej wykonywana jest operacja pobierania podłańcuchów z łańcucha niż kodów poszczególnych znaków. Aby wykonać to działanie, należy w nawiasach kwadratowych umieścić dwa operandy oddzielone przecinkiem. Pierwszy z nich określa indeks (może być ujemny), a drugi długość (nie może być ujemna). Wynikiem tej operacji jest podłańcuch zaczynający

sie od znaku pod podanym indeksem i zawierający liczbę znaków określona przez drugą z wartości:

```
s = "hello"
s[0,2]           # "he".
s[-1,1]          # ":" zwraca łańcuch, nie kod znaku ?o.
s[0,0]           # ":" podłańcuch o długości zero jest zawsze pusty.
s[0,10]          # "hello": zwraca wszystkie dostępne znaki.
s[s.length,1]    # ":" bezpośrednio za końcem jest pusty łańcuch.
s[s.length+1,1]  # nil: odczyt za tym miejscem jest błędem.
s[0,-1]          # nil: ujemne długości są pozbawione sensu.
```

Przypisanie łańcucha do łańcucha indeksowanego w ten sposób powoduje zastąpienie określonego podłańcucha nowym łańcuchem. Jeśli po prawej stronie jest pusty łańcuch, wykonywane jest usuwanie; jeżeli po lewej stronie długość jest ustawiona na zero, wykonywane jest wstawianie:

```
s = "hello"
s[0,1] = "H"           # Zamiana pierwszej litery na wielką.
s[s.length,0] = " world" # Dodanie słowa na końcu łańcucha.
s[5,0] = ""            # Wstawienie przecinka bez usuwania czegokolwiek.
s[5,6] = ""            # Usunięcie bez wstawiania; s == "Hello!".
```

Innym sposobem na wydobycie, wstawienie, usunięcie lub podmianę podłańcucha jest indeksowanie łańcucha za pomocą obiektu Range (zakres). Zakresy zostały szczegółowo opisane dalej w podrozdziale 3.5. Na bieżące potrzeby wystarczy wiedzieć, że obiekt typu Range to dwie liczby całkowite rozdzielone kropkami. Kiedy do indeksowania łańcucha zostaje użyty zakres, wartością zwrotną jest podłańcuch, którego znaki mieszczą się w tym zakresie:

```
s = "hello"
s[2..3]           # "ll": znaki numer 2 i 3.
s[-3..-1]          # "lo": ujemne indeksy także działają.
s[0..0]           # "h": zakres jednoznakowy.
s[0..0]           # ":" pusty zakres.
s[2..1]           # ":" kolejny pusty zakres.
s[7..10]          # nil: zakres poza granicami łańcucha.
s[-2..-1] = "p!"   # Zamiana: s zamienia się na "help!".
s[0..0] = "Please " # Wstawianie: s zamienia się na "Please help!".
s[6..10] = ""       # Usuwanie: s zamienia się na "Please!".
```

Nie należy mylić techniki indeksowania polegającej na użyciu dwóch liczb całkowitych oddzielonych przecinkiem z obiektem klasy Range. Mimo iż w obu przypadkach używane są dwie liczby całkowite, jest między nimi istotna różnica: sposób z przecinkiem pozwala określić indeks i długość, a obiekt klasy Range określa dwa indeksy.

Możliwe jest także indeksowanie łańcucha przy użyciu łańcucha. W takim przypadku wartością zwrotną jest pierwszy podłańcuch łańcucha docelowego pasujący do łańcucha indeksowego lub wartość nil, jeśli nie ma pasującego łańcucha. Ten rodzaj indeksowania łańcuchów bywa przydatny tylko po lewej stronie instrukcji przypisania, kiedy trzeba zastąpić dopasowany łańcuch jakimś innym łańcuchem:

```
s = "hello"        # Na początku mamy łańcuch "hello".
while(s["l"])       # Każde wystąpienie podłańcucha "l"
    s["l"] = "L";   # zastępowane jest łańcuchem "L".
end                # Powstał łańcuch "heLLo".
```

W końcu łańcuch można indeksować, wykorzystując wyrażenia regularne (obiekty wyrażeń regularnych zostały opisane w podrozdziale 9.2). Wynikiem tej operacji jest pierwszy podłańcuch łańcucha, który pasuje do podanego wzorca. Ten sposób indeksowania jest również najbardziej przydatny po lewej stronie instrukcji przypisania:

```
s[/[aeiouy]/] = '*'      # Zastąpienie pierwszej samogłoski gwiazdką.
```

3.2.5. Iteracja przez łańcuchy

W języku Ruby 1.8 klasa `String` udostępnia metodę o nazwie `each`, która iteruje przez łańcuch wiersz po wierszu. Klasa ta zawiera metody modułu `Enumerable`; używając ich, można przetwarzać wiersze tego łańcucha. W Ruby 1.8 można używać metody iteracyjnej `each_byte` przechodzącej przez wszystkie bajty łańcucha. Metoda ta ma jednak niewielką przewagę nad operatorem `[]`, ponieważ swobodny dostęp do bajtów w Ruby 1.8 jest tak samo szybki jak dostęp sekwencyjny.

Sytuacja jest radykalnie inna w Ruby 1.9. Usunięto metodę `each`, a klasa `String` nie implementuje modułu `Enumerable`. Zamiast metody `each` dostępne są trzy iteratory łańcuchowe o jasnych nazwach: `each_byte` — iteruje sekwencyjnie przez poszczególne bajty składające się na łańcuch, `each_char` — iteruje przez znaki oraz `each_line` — iteruje przez wiersze. Aby przetworzyć cały łańcuch znak po znaku, efektywniejszym rozwiązaniem może być użycie metody `each_char` zamiast operatora `[]` i indeksów znaków:

```
s = "¥1000"
s.each_char { |x| print "#{x} " }           # Drukujes "¥ 1 0 0 0". Ruby 1.9.
0.upto(s.size-1) { |i| print "#{s[i]}"}   # Powolna w przypadku wielobajtowych znaków.
```

3.2.6. Kodowanie łańcuchów a znaki wielobajtowe

Pomiędzy łańcuchami w Ruby 1.8 i 1.9 są fundamentalne różnice:

- W Ruby 1.8 łańcuch jest ciągiem bajtów. Kiedy łańcuch reprezentuje tekst (zamiast danych binarnych), każdy jego bajt jest traktowany jako reprezentacja jednego znaku ASCII. W Ruby 1.8 poszczególne elementy łańcucha nie są znakami, a liczbami — rzeczywistymi wartościami bajtów lub kodami znaków.
- W Ruby 1.9 łańcuchy są prawdziwymi ciągami znaków, które nie muszą należeć do zestawu ASCII. Poszczególne elementy łańcucha są znakami — reprezentowanymi jako łańcuchy o długości 1 — a nie całkowitoliczbowymi kodami znaków. Każdy łańcuch posiada kodowanie określające zależność między bajtami w nim zawartymi a znakami przez nie reprezentowanymi. W systemach kodowania, takich jak UTF-8 Unicode, znaki są reprezentowane przez różne ilości bajtów, dlatego nie istnieje w nich zależność 1 do 1 (a nawet 2 do 1) pomiędzy bajtami i znakami.

Poniższe podrozdziały opisują kodowanie łańcuchów w języku Ruby 1.9 oraz podstawową obsługę znaków wielobajtowych w Ruby 1.8 przy użyciu biblioteki `jcode`.

3.2.6.1. Znaki wielobajtowe w Ruby 1.9

W Ruby 1.9 w klasie `String` dodano obsługę znaków wielobajtowych. Mimo iż jest ona największą zmianą w tej wersji języka, innowacja ta nie jest od razu widoczna — kod używający znaków wielobajtowych po prostu działa. Ważne jest jednak, aby zrozumieć, dlaczego on działa. Temu właśnie został poświęcony ten podrozdział.

Jeśli łańcuch zawiera znaki wielobajtowe, liczba bajtów nie odpowiada liczbie znaków. W Ruby 1.9 metody `length` i `size` zwracają liczbę znaków w łańcuchu, a metoda `bytesize` zwraca liczbę bajtów. Operatory `[]` oraz `[]=` pozwalają na sprawdzenie oraz ustawienie znaków łańcucha, a nowe metody `getbyte` oraz `setbyte` — na sprawdzenie i ustawienie poszczególnych bajtów (jednak rzadko jest to konieczne):

```
# -*- coding: utf-8 -*- # Znaki UTF-8 Unicode.
# Literal łańcuchowy zawierający wielobajtowy znak mnożenia.
s = "2×2=4"
# Łąćuch składa się z sześciu bajtów kodujących pięć znaków.
s.bytesize                                # => 6.
s.bytesize.times { |i| print s.getbyte(i), " " } # Wyświetla "50 195 151 50 61 52".
s.length                                    # => 5.
s.length.times { |i| print s[i], " " }        # Wyświetla "2 Ą— 2 = 4".
s.setbyte(5, s.getbyte(5)+1);                 # s ma teraz wartość "2Ā—2=5".
```

Należy zauważyc, że w pierwszym wierszu powyższego kodu znajduje się komentarz ustanawiający kodowanie źródła (zobacz podrozdział 2.4.1) na UTF-8. Bez niego interpreter nie wiedziałby, jak przetworzyć sekwencję bajtów na sekwencję znaków.

Kiedy łańcuch zawiera znaki zakodowane przy użyciu różnych liczb bajtów, nie ma możliwości bezpośredniego znalezienia znaku w łańcuchu na podstawie indeksu. Na przykład w powyższym łańcuchu drugi znak zaczyna się od drugiego bajta. Ale trzeci znak zaczyna się już od bajta czwartego. Oznacza to, że nie można zakładać, iż swobodny dostęp do dowolnego znaku w łańcuchu jest szybki. Jeśli dostęp do znaku lub podłańcucha w wielobajtowym łańcuchu jest uzyskiwany za pomocą operatora [], tak jak w powyższym przykładzie, Ruby musi przejść kolejno przez cały łańcuch w celu znalezienia indeksu żądanego znaku. W związku z tym łańcuchy powinno przetwarzac się w miarę możliwości przy użyciu algorytmów sekwencyjnych. To znaczy zamiast wielokrotnych wywołań operatora [] należy używać metody iteracyjnej each_char, jeśli jest to możliwe. Z drugiej strony często nie ma to żadnego znaczenia. Ruby optymalizuje te przypadki, które da się zoptymalizować. Dlatego jeśli łańcuch w całości składa się ze znaków jednobajtowych, swobodny dostęp do tych znaków będzie efektywny. Aby spróbować optymalizacji we własnym zakresie, można użyć metody obiektowej ascii_only?, która sprawdza, czy łańcuch składa się wyłącznie z siedmiobajtowych znaków ASCII.

Klasa String w Ruby 1.9 udostępnia metodę encoding zwracającą kodowanie łańcucha (wartością zwrotną jest obiekt typu Encoding, który został opisany niżej):

```
# -*- coding: utf-8 -*-
s = "2×2=4"          # Zauważ wielobajtowy znak mnożenia.
s.encoding           # => <Encoding: UTF-8>.
t = "2+2=4"          # Wszystkie znaki mieszczą się w podzbiорze ASCII UTF-8.
t.encoding           # => <Encoding: ASCII-8BIT>.
```

Kodowanie literalu łańcuchowego odpowiada kodowaniu źródła pliku, w którym łańcuch ten się znajduje. Jednak kodowanie to nie zawsze jest takie samo jak źródłowe. Jeśli na przykład literał łańcuchowy zawiera same siedmiobajtowe znaki ASCII, metoda encoding zwróci ASCII, nawet w przypadku gdy kodowanie źródła to UTF-8 (nadzbiór ASCII). Ta optymalizacja informuje metody łańcuchowe, że wszystkie znaki w łańcuchu mają długość jednego bajta. Dodatkowo jeżeli literał łańcuchowy zawiera sekwencje specjalne \u, jego kodowanie będzie określone jako UTF-8, nawet gdy kodowanie źródła jest inne.

Niektóre działania na łańcuchach, takie jak konkatenacja czy dopasowywanie wzorców, wymagają, aby oba łańcuchy (lub łańcuch i wyrażenie regularne) miały zgodne ze sobą kodowania. Na przykład w wyniku konkatenacji łańcucha ASCII z łańcuchem UTF-8 powstanie łańcuch UTF-8. Nie jest jednak możliwe połączenie łańcuchów UTF-8 i SJIS — kodowania te nie są ze sobą zgodne i zostanie zgłoszony wyjątek. Do sprawdzania zgodności dwóch łańcuchów (lub łańcucha i wyrażenia regularnego) służy metoda klasowa Encoding.compatible?. Jeśli kodowania obu argumentów są ze sobą zgodne, metoda zwraca to, które jest nadzbiorem drugiego. Jeśli kodowania są niezgodne, zwrócona zostaje wartość nil.

Kodowanie ASCII i BINARY

Pokazane wcześniej kodowanie ASCII-8BIT jest nazwą stosowaną w Ruby 1.9 do określenia starego kodowania używanego w Ruby 1.8. Jest to zestaw znaków ASCII bez żadnych ograniczeń dotyczących używania znaków niedrukowalnych i sterujących. W tym kodowaniu jeden bajt to zawsze jeden znak, a łańcuchy mogą przechowywać zarówno dane binarne, jak i znakowe.

Niektóre metody Ruby 1.9 wymagają podania nazwy kodowania (lub obiektu Encoding, który jest opisany niżej). Można podać nazwę ASCII-8BIT lub jej alias BINARY. Może się to wydawać zaskakujące, ale to prawda — w języku Ruby ciąg bajtów bez kodowania (BINARY) jest tym samym co ciąg ośmiobitowych znaków ASCII.

Ruby obsługuje także kodowanie o nazwie US-ASCII będące siedmiobitowym ASCII. Od ASCII-8BIT różni się ono tym, że nie pozwala na użycie bajtów z ustawionym ósmym bitem. Nazwa kodowania ASCII jest aliasem dla US-ASCII.

Kodowanie łańcucha można ustawić jawnie za pomocą metody `force_encoding`. Pozwala ona poinformować Ruby, jak należy traktować jako znaki łańcuch bajtów (najczęściej wczytyanych ze strumienia wejściowego). Inna sytuacja to łańcuch znaków wielobajtowych, których poszczególne bajty trzeba indeksować, stosując operator `[]`:

```
text = stream.readline.force_encoding("utf-8")
bytes = text.dup.force_encoding(nil) # Kodowanie nil oznacza binarne.
```

Metoda `force_encoding` nie tworzy kopii swojego adresata. Modyfikuje kodowanie łańcucha i zwraca ten łańcuch. Nie konwertuje żadnych znaków — bajty łańcucha pozostają niezmienione, zmienia się tylko sposób ich interpretacji przez Ruby. Jak widać powyżej, argumentem metody `force_encoding` może być nazwa kodowania lub wartość `nil` oznaczająca kodowanie binarne. Kodowanie można także określić, przekazując obiekt typu `Encoding`.

Metoda `force_encoding` nie wykonuje walidacji. Nie sprawdza, czy bajty łańcucha reprezentują prawidłowy ciąg znaków w określonym kodowaniu. Do walidacji służy metoda `valid_encoding?`. Niniejsza metoda obiektowa nie pobiera żadnych argumentów; sprawdza natomiast, czy bajty w łańcuchu można zinterpretować jako prawidłowy szereg znaków w danym kodowaniu:

```
s = "\xa4".force_encoding("utf-8") # To nie jest prawidłowy łańcuch UTF-8.
s.valid_encoding? # => false.
```

Metoda `encode` (i jej mutacyjna wersja `encode!`) łańcucha działa inaczej niż `force_encoding`. Zwraca łańcuch reprezentujący ten sam ciąg znaków co jej adresat, ale przy użyciu innego kodowania. Aby zmienić kodowanie takiego łańcucha, metoda `encode` musi zmienić bajty składające się na ten łańcuch. Oto przykład:

```
# -*- coding: utf-8 -*-
euro1 = "\u20AC" # Na początku jest znak Unicode Euro.
puts euro1 # Drukuje "€".
euro1.encoding # => <Encoding:UTF-8>.
euro1.bytesize # => 3.
euro2 = euro1.encode("iso-8859-15") # Zmiana kodowania na Latin-15.
puts euro2.inspect # Drukuje "\xA4".
euro2.encoding # => <Encoding:iso-8859-15>.
euro2.bytesize # => 1.
euro3 = euro2.encode("utf-8") # Zmiana kodowania z powrotem na UTF-8.
euro1 == euro3 # => true.
```

Pamiętaj, że użycie metody `encode` nie powinno być zbyt często potrzebne. Najczęściej zmienia się kodowanie łańcuchów przed zapisaniem ich w pliku lub wysłaniem za pośrednictwem sieci. W podrozdziale 9.7.2 opisane są klasy wejścia-wyjścia dokonujące automatycznej konwersji kodowania tekstu w czasie jego wysyłania na wyjście.

Jeśli łańcuch, na rzecz którego wywoływana jest metoda `encode`, zawiera niezakodowane znaki, należy przed zmianą jego kodowania na inne określić, według jakiego kodowania bajty mają zostać zinterpretowane. W tym celu do metody `encode` należy przekazać dwa argumenty. Pierwszy z nich określa kodowanie docelowe, a drugi kodowanie bieżące łańcucha. Na przykład:

```
# Interpretacja bajta jako punkt kodowy iso-8859-15 i zmiana kodowania na UTF-8.  
byte = "\xA4"  
char = byte.encode("utf-8", "iso-8859-15")
```

Oznacza to, że poniższe dwa wiersze kodu dają taki sam efekt:

```
text = bytes.encode(to, from)  
text = bytes.dup.force_encoding(from).encode(to)
```

Kodowania znaków różnią się nie tylko sposobem odwzorowywania bajtów na znaki, ale również zestawem reprezentowanych znaków. Unicode (nazywany też UCS — Universal Character Set) próbuje uwzględnić wszystkie znaki, ale kodowania nieoparte na nim mogą reprezentować tylko podzbiór znaków. Dlatego niemożliwe jest przekonwertowanie łańcucha UTF-8 na EUC-JP (na przykład). Znaki Unicode, które nie są ani łacińskie, ani japońskie, nie mogą być poddane translacji.

Jeśli metoda `encode` lub `encode!` napotka znak, którego kodowania nie może zmienić, zgłasza wyjątek:

```
# W zestawie iso-8859-1 nie ma znaku Euro, a więc kod ten spowoduje wyjątek.  
\u20AC.encode("iso-8859-1")
```

Metody `encode` oraz `encode!` przyjmują tablicę asocjacyjną opcji kodowania jako ostatni argument. W chwili pisania tej książki jedyną zdefiniowaną opcją była opcja `:invalid`, a jedną zdefiniowaną wartością dla tego klucza — `:ignore`. Gdy zostanie zaimplementowanych więcej opcji, odpowiednie informacje można będzie uzyskać, wywołując `ri String.encode`.

3.2.6.2. Klasa Encoding

Klasa `Encoding` w Ruby 1.9 reprezentuje kodowanie znaków. Obiekty klasy `Encoding` działają jak nieprzejrzyste identyfikatory kodowania i nie posiadają zbyt wielu własnych metod. Metoda `name` zwraca nazwę kodowania. Metoda `to_s` jest synonimem metody `name`, a `inspect` konwertuje obiekt klasy `Encoding` na łańcuch bardziej szczegółowo niż metoda `name`.

W Ruby każde obsługiwane wbudowane kodowanie ma swoją stałą. Jej użycie jest najprostszym sposobem określenia kodowania w programie. Do predefiniowanych stałych należą:

<code>Encoding::ASCII_8BIT</code>	<code># Także ::BINARY.</code>
<code>Encoding::UTF_8</code>	<code># Znaki Unicode zakodowane w UTF-8.</code>
<code>Encoding::EUC_JP</code>	<code># Japońskie znaki EUC.</code>
<code>Encoding::SHIFT_JIS</code>	<code># Japońskie: także ::SJIS, ::WINDOWS_31J, ::CP932.</code>

Należy zauważyc, że stałe muszą być zapisane wielkimi literami, a myślniki w nazwach kodowania muszą być zamienione na znaki podkreślenia. Ruby obsługuje także kodowanie US-ASCII, europejskie kodowania ISO-8859-1 do ISO-8859-15 oraz Unicode UTF-16 i UTF-32 w wariantach big-endian i little-endian.

Aby utworzyć obiekt klasy `Encoding` z łańcucha reprezentującego nazwę kodowania, należy użyć metody fabrycznej `Encoding.find`:

```
encoding = Encoding.find("utf-8")
```

Metoda `Encoding.find` powoduje w razie potrzeby dynamiczne załadowanie określonego kodowania. Przyjmuje ona nazwy kodowania zapisane zarówno małymi, jak i wielkimi literami. Aby uzyskać nazwę kodowania w postaci łańcucha, należy wywołać metodę `name` obiektu klasy `Encoding`.

Metoda `Encoding.list` zwraca tablicę zawierającą wszystkie dostępne obiekty `Encoding`. Metoda `Encoding.name_list` zwraca tablicę nazw (obiektów `String`) wszystkich dostępnych kodowań. Wiele kodowań posiada kilka nazw, a metoda `Encoding.aliases` zwraca tablicę asocjacyjną odwzorowującą aliasy na oficjalne nazwy kodowania. Tablica zwracana przez metodę `Encoding.name_list` uwzględnia aliasy, które zwraca metoda `Encoding.aliases`.

Metoda `Encoding.default_external` zwraca obiekt klasy `Encoding` reprezentujący domyślne kodowanie zewnętrzne (zobacz podrozdział 2.4.2). Aby sprawdzić kodowanie bieżącej lokalizacji, należy wywołać metodę `Encoding.locale_charmap` i zwrócony przez nią łańcuch przekazać do metody `Encoding.find`.

Większość metod, które pobierają na wejściu obiekt typu `Encoding`, przyjmuje także nazwę kodowania (bez uwzględniania wielkości liter, np. `ascii`, `binary`, `utf-8`, `euc-jp` lub `sjis`) zamiast obiektu klasy `Encoding`.

3.2.6.3. Znaki wielobajtowe w Ruby 1.8

Normalnie Ruby 1.8 traktuje wszystkie łańcuchy jako ciągi ośmiobitowych bajtów. Za podstawową obsługę znaków wielobajtowych (używając zestawów UTF-8, EUC lub SJIS) odpowiada moduł `jcode` w bibliotece standardowej.

Aby jej użyć, należy dołączyć za pomocą metody `require` moduł `jcode` i ustawić zmienną globalną `$KCODE` na kodowanie używanych znaków wielobajtowych (można też użyć opcji wiersza poleceń `-K` podczas uruchamiania interpretera Ruby). Biblioteka `jcode` udostępnia nową metodę `jlength` dla obiektów typu `String` — zwraca liczbę znaków w łańcuchu, a nie liczbę bajtów. Istniejące metody `length` i `size` w Ruby 1.8 pozostają niezmienione — zwracają długość łańcuchów w bajtach.

Biblioteka `jcode` nie modyfikuje operatora indeksującego tablice na łańcuchach oraz nie zezwala na swobodny dostęp do znaków składających się na wielobajtowy łańcuch. Definiuje natomiast nową metodę iteracyjną o nazwie `each_char`, która działa jak standardowa metoda `each_byte`, tyle że przekazuje każdy znak łańcucha (jako łańcuch, a nie jako kod znaku) do dostarczonego bloku kodu:

```
$KCODE = "u"          # Określ kodowanie UTF-8 lub uruchom Ruby z opcją -Ku.
require "jcode"        # Załadowanie obsługi znaków wielobajtowych.
mb = "2\303\2272=4"  # Łańcuch "2×2=4" ze znakiem mnożenia Unicode.
mb.length              # => 6: ten łańcuch składa się z sześciu bajtów,
mb.jlength             # => 5: ale tylko pięciu liter.
mb.mbchar?             # => 1: położenie pierwszego znaku wielobajtowego lub nil.
mb.each_byte do |c|   # Iteracja przez bajty łańcucha.
  print c, " "         # c jest typu Fixnum.
end                   # Drukuję "50 195 151 50 61 52 ".
mb.each_char do |c|  # Iteracja przez znaki łańcucha.
  print c, " "         # c jest łańcuchem z jlength 1.
end                   # Drukuję "2 × 2 = 4 ".
```

Biblioteka `jcode` zawiera także zmodyfikowane wersje kilku istniejących metod `String`, jak `chop`, `delete` i `tr`, które działają na łańcuchach wielobajtowych.

3.3. Tablice

Tablica to sekwencja wartości, do których dostęp można uzyskać, określając ich położenie w szeregu, czyli za pomocą indeksów. Pierwsza wartość w tablicy ma indeks 0. Metody `size` i `length` zwracają liczbę elementów w tablicy. Ostatni element znajduje się pod indeksem `size-1`. Ujemne wartości indeksów są odliczane od końca tablicy. Zatem do ostatniego elementu tablicy można uzyskać dostęp za pomocą indeksu `-1`. Przedostatni element ma indeks `-2` itd. Próba odczytu elementu spoza tablicy (`size=< indeks <-size`) kończy się zwróceniem wartości `nil` bez zgłoszenia wyjątku.

Tablice w Ruby nie mają określonego typu i można je modyfikować. Elementy takiej tablicy nie muszą należeć do jednej klasy i można je zmienić w dowolnym momencie. Ponadto tablice mogą dynamicznie zmieniać rozmiary. W wyniku dodawania nowych obiektów powiększają się zgodnie z zapotrzebowaniem. Jeśli do elementu znajdującego się poza końcem tablicy zostanie przypisana wartość, tablica zostanie automatycznie rozszerzona i dopełniona wartościami `nil` (będącym jest natomiast przypisanie wartości do elementu przed początkiem tablicy).

Literał tablicowy to lista wartości oddzielonych przecinkami otoczona nawiasami kwadratowymi:

```
[1, 2, 3]          # Tablica przechowująca trzy obiekty typu Fixnum.  
[-10...0, 0..10,] # Tablica dwóch zakresów. Dozwolone jest stosowanie przecinka na końcu.  
[[1,2],[3,4],[5]] # Tablica tablic.  
[x+y, x-y, x*y]  # Elementy tablicy mogą być dowolnymi wyrażeniami.  
[]                # Pusta tablica ma rozmiar 0.
```

W Ruby jest dostępna specjalna składnia dla literałów tablicowych, których elementy są krótkimi łańcuchami niezawierającymi spacji:

```
words = %w{to jest test}    # To samo co: ['to', 'jest', 'test'].  
open = %w| ( { < |         # To samo co: ['(', '{', '<', '|'.  
white = %w(\s \t \r \n)     # To samo co: ["\s", "\t", "\r", "\n"].
```

Łańcuchy `%w` i `%W` wprowadzają literały tablicowe, tak jak `%q` i `%Q` literały łańcuchowe. Zasady dotyczące ograniczników `%w` i `%W` są takie same jak dotyczące `%q` i `%Q`. W obrębie tych ograniczników łańcuchy będące elementami tablicy nie muszą być otoczone żadnymi cudzysłowniami, a do rozdzielania elementów nie są potrzebne przecinki. Elementy tablicy są rozdzielane białymi znakami.

Tablice można także tworzyć za pomocą konstruktora `Array.new`. Przy użyciu tej metody można zainicjować w programie elementy tablicy:

```
empty = Array.new        # []: zwraca nową pustą tablicę.  
nils = Array.new(3)      # [nil, nil, nil]: nowa tablica z trzema elementami nil.  
zeros = Array.new(4, 0)   # [0, 0, 0, 0]: nowa tablica z czterema elementami 0.  
copy = Array.new(nil)    # Utworzenie nowej kopii istniejącej tablicy.  
count = Array.new(3) { |i| i+1} # [1,2,3]: Trzy elementy obliczone z indeksu.
```

Aby pobrać wartość wybranego elementu tablicy, należy w nawiasach kwadratowych umieścić odpowiednią liczbę całkowitą:

```
a = [0, 1, 4, 9, 16] # Tablica przechowująca potęgi kwadratowe indeksów.  
a[0]                  # Pierwszy element ma wartość 0.  
a[-1]                 # Ostatni element ma wartość 16.
```

```

a[-2]           # Przedostatni element ma wartość 9.
a[a.size-1]     # Inny sposób na sprawdzenie wartości ostatniego elementu.
a[-a.size]      # Inny sposób na sprawdzenie wartości pierwszego elementu.
a[8]            # Sprawdzenie wartości poza tablicą zwraca wartość nil.
a[-8]           # Sprawdzenie wartości przed tablicą również zwraca wartość nil.

```

Wszystkie przedstawione wyżej wyrażenia, z wyjątkiem ostatniego, mogą zostać użyte po lewej stronie instrukcji przypisania:

```

a[0] = "zero"    # a wynosi ["zero", 1, 4, 9, 16].
a[-1] = 1..16    # a wynosi ["zero", 1, 4, 9, 1..16].
a[8] = 64        # a wynosi ["zero", 1, 4, 9, 1..16, nil, nil, nil, 64].
a[-10] = 100     # Błąd: nie można wykonać przypisania przed początkiem tablicy.

```

Tablice, podobnie jak łańcuchy, można indeksować za pomocą dwóch liczb całkowitych reprezentujących indeks początkowy i liczbę elementów lub obiektu typu Range. W każdym z tych przypadków wyrażenie zwraca określzoną podtablicę:

```

a = ('a'..'e').to_a    # Zakres przekonwertowany na ['a', 'b', 'c', 'd', 'e'].
a[0,0]                 # []: mniejsza podtablica ma zero elementów.
a[1,1]                 # ['b']: tablica jednoelementowa.
a[-2,2]                # ['d','e']: dwa ostatnie elementy tablicy.
a[0..2]                # ['a', 'b', 'c']: trzy pierwsze elementy.
a[-2...-1]              # ['d','e']: dwa ostatnie elementy tablicy.
a[0...-1]               # ['a', 'b', 'c', 'd']: wszystkie elementy oprócz ostatniego.

```

Podtablica użyta po lewej stronie instrukcji przypisania zostaje zastąpiona elementami tablicy po prawej stronie. W ten sposób można także wstawiać i usuwać elementy:

```

a[0,2] = ['A', 'B']      # Tablica A zawiera elementy ['A', 'B', 'c', 'd', 'e'].
a[2...5]=['C', 'D', 'E']  # Tablica A zawiera elementy ['A', 'B', 'C', 'D', 'E'].
a[0,0] = [1, 2, 3]        # Wstawienie elementów na początek tablicy a.
a[0..2] = []              # Usunięcie tych elementów.
a[-1,1] = ['Z']           # Zamiana ostatniego elementu na inny.
a[-1,1] = 'Z'             # W przypadku pojedynczych elementów tablica jest rozwiązaniem opcjonalnym.
a[-2,2] = nil             # Usunięcie dwóch ostatnich elementów w Ruby 1.8. Zastąpienie wartościami nil w 1.9.

```

Poza operatorem nawiasów kwadratowych służącym do indeksowania tablicy klasa Array udostępnia kilka innych przydatnych operatorów. Do konkatenacji tablic służy operator +:

```

a = [1, 2, 3] + [4, 5]    # [1, 2, 3, 4, 5].
a = a + [[6, 7, 8]]       # [1, 2, 3, 4, 5, [6, 7, 8]].
a = a + 9                 # Błąd: po prawej stronie musi być tablica.

```

Operator + tworzy nową tablicę zawierającą elementy obu swoich operandów. Aby dodać element na końcu istniejącej tablicy, można użyć operatora <<. W celu dodania większej liczby elementów można użyć metody concat:

```

a = []                  # Na początku tablica jest pusta.
a << 1                  # Tablica a = [1].
a << 2 << 3            # Tablica a = [1, 2, 3].
a << [4, 5, 6]          # Tablica a = [1, 2, 3, [4, 5, 6]].
a.concat [7,8]           # Tablica a = [1, 2, 3, [4, 5, 6], 7, 8].

```

Operator - odejmuje jedną tablicę od drugiej. Najpierw tworzy kopię tablicy znajdującej się po lewej stronie, a następnie usuwa z niej wszystkie elementy, które można znaleźć w tablicy po prawej stronie:

```
['a', 'b', 'c', 'b', 'a'] - ['b', 'c', 'd']    # ['a', 'a'].
```

Klasa Array, podobnie jak String, używa operatora mnożenia do zwielokrotniania:

```
a = [0] * 8    # [0, 0, 0, 0, 0, 0, 0, 0].
```

Klasa `Array` pożycza operatory logiczne `|` i `&`, które oznaczają sumę zbiorów i przecięcie zbiorów. Operator `|` łączy swoje argumenty, a następnie usuwa duplikaty. Operator `&` zwraca tablicę zawierającą elementy obecne w obu operandach. Zwrócona tablica nie zawiera żadnych duplikatów:

```
a = [1, 1, 2, 2, 3, 3, 4]
b = [5, 5, 4, 4, 3, 3, 2]
a | b      # [1, 2, 3, 4, 5]: duplikaty są usuwane.
b | a      # [5, 4, 3, 2, 1]: te same elementy, tylko w innej kolejności.
a & b      #[2, 3, 4].
b & a      #[4, 3, 2].
```

Należy zauważać, że operatory te nie są przemienne — `a|b` to nie to samo co `b|a`. Są one bardziej przydatne przy ignorowaniu kolejności elementów i traktowaniu tablic jako zbiorów nieuporządkowanych. Istotne jest również, że algorytmy, według których obliczane są suma i przecięcie zbiorów, nie są określone, przez co nie wiadomo, jaka będzie kolejność elementów w zwróconych tablicach.

Klasa `Array` udostępnia dużo przydatnych metod. Poniżej zostanie opisana tylko metoda iteracyjna `each`, która odwiedza elementy tablicy:

```
a = ('A'..'Z').to_a      # Na początku jest tablica liter.
a.each { |x| print x }  # Wydruk alfabetu po jednej literze.
```

Inne metody klasy `Array`, które warto sprawdzić, to: `clear`, `compact!`, `delete_if`, `each_index`, `empty?`, `fill`, `flatten!`, `include?`, `index`, `join`, `pop`, `push`, `reverse`, `reverse_each`, `rindex`, `shift`, `sort`, `sort!`, `uniq!` oraz `unshift`.

Do tablic wracamy jeszcze przy temacie przypisywania równoległego w podrozdziale 4.5.5 i wywoływanie metod w rozdziale 6. API klasy `Array` zostało szczegółowo opisane w podrozdziale 9.5.2.

3.4. Tablice asocjacyjne

Tablica asocjacyjna (ang. *hash*) to struktura danych przechowująca zbiór obiektów zwanych **kluczami** i skojarzone z nimi wartości. Tablice asocjacyjne nazywane są także **mapami** (ang. *map*), ponieważ odwzorowują klucze na wartości, oraz **haszami**. Oto przykład tablicy asocjacyjnej:

```
# Niniejsza tablica asocjacyjna kojarzy nazwy cyfr z cyframi.
numbers = Hash.new      # Utworzenie nowego pustego obiektu typu Hash.
numbers["jeden"] = 1    # Powiązanie łańcucha "jeden" z obiektem typu Fixnum 1.
numbers["dwa"] = 2      # Użycie notacji tablicowej.
numbers["trzy"] = 3
sum = numbers["jeden"] + numbers["dwa"]  # W ten sposób pobiera się wartości.
```

Niniejsze wprowadzenie do tablic asocjacyjnych w języku Ruby objaśnia składnię literałów haszowych (tablic asocjacyjnych) oraz przedstawia wymogi, które obiekt musi spełnić, aby mógł być użyty jako klucz.Więcej informacji na temat API klasy `Hash` znajduje się w podrozdziale 9.5.3.

3.4.1. Literaty haszowe

Literał haszowy ma postać listy par klucz-wartość rozdzielonych przecinkami i zamkniętych w nawiasach klamrowych. Klucze i wartości są rozdzielane dwuznakowymi strzałkami `=>`.

Utworzony wcześniej obiekt typu Hash można utworzyć także za pomocą poniższego literału:

```
numbers = { "jeden" => 1, "dwa" => 2, "trzy" => 3 }
```

Ogólnie jako klucze w tablicach asocjacyjnych efektywniejsze od łańcuchów są obiekty typu Symbol:

```
numbers = { :jeden => 1, :dwa => 2, :trzy => 3 }
```

Symbole to niemodyfikowalne łańcuchy wewnętrzne zapisywane jako identyfikatory z przedrostkiem w postaci dwukropka. Więcej szczegółów na ich temat znajduje się w podrozdziale 3.6.

Ruby 1.8 pozwala na używanie przecinków zamiast strzałek, ale ta wycofywana składnia nie jest już dozwolona w Ruby 1.9:

```
numbers = { :jeden, 1, :dwa, 2, :trzy, 3 } # To samo, ale trudniejsze do czytania.
```

Zarówno Ruby 1.8, jak i 1.9 pozwalają na wstawienie końcowego przecinka na końcu listy par klucz-wartość:

```
numbers = { :one => 1, :two => 2, } # Dodatkowy przecinek jest ignorowany.
```

Kiedy klucze są symbolami, w Ruby 1.9 można używać bardzo zwięzlej składni. Dwukropki przechodzą na koniec klucza i zastępuje strzałkę³:

```
numbers = { jeden: 1, dwa: 2, trzy: 3 }
```

Należy pamiętać, że pomiędzy identyfikatorem klucza a dwukropkiem nie może być spacji.

3.4.2. Kody mieszające, równość i klucze modyfikowalne

Nie jest zaskakujące, że tablice asocjacyjne w języku Ruby są zaimplementowane za pomocą struktury danych o nazwie **tablicy mieszającej** (ang. *hash table*). Obiekty służące jako klucze w tablicy mieszającej muszą dysponować metodą o nazwie hash, która zwraca **kod mieszający** (ang. *hashcode*) typu Fixnum tych obiektów. Aby dwa klucze były identyczne, muszą mieć jednakowe kody mieszające. Nieidentyczne klucze również mogą mieć taki sam kod mieszający, ale tablice mieszające są najbardziej efektywne przy niewielkiej liczbie duplikatów kodów mieszających.

Do porównywania kluczy w klasie Hash służy metoda eql?. W większości klas Ruby metoda eql? działa tak samo jak operator == (szczegóły na ten temat znajdują się w podrozdziale 3.8.5). Jeśli w nowo utworzonej klasie zostanie przesłonięta metoda eql?, konieczne jest też przesłonięcie metody hash. W przeciwnym wypadku egzemplarze tej klasy nie będą działać jak klucze w tablicach asocjacyjnych (przykłady pisania metody hash znajdują się w rozdziale 7.).

Jeżeli w zdefiniowanej klasie nie zostanie przesłonięta metoda eql?, to jej obiekty używane jako klucze będą porównywane pod względem identyczności. Dwa różne egzemplarze klasy mają różne kody mieszające, nawet gdy zawierają taką samą treść. W takim przypadku właściwa jest domyślna metoda hash zwracająca unikatowy identyfikator object_id obiektu.

Należy zauważyć, że obiekty modyfikowalne jako klucze mogą sprawiać problemy. Zmiana treści obiektu zazwyczaj pociąga za sobą zmianę jego kodu mieszającego. Jeśli obiekt użyty jako klucz zostanie zmodyfikowany, wewnętrzna tablica mieszająca zostanie uszkodzona i nie będzie działać poprawnie.

³ Składnia ta jest podobna do składni używanej przez obiekty w języku JavaScript.

Ponieważ łańcuchy są modyfikowalne, ale powszechnie używa się ich w roli kluczy, Ruby traktuje je w wyjątkowy sposób, robiąc prywatną kopię każdego z nich. Jest to jednak specjalny przypadek. Używając innych typów modyfikowalnych obiektów jako kluczy, należy zachować szczególną ostrożność. Rozważ tworzenie prywatnej kopii lub wywołanie metody `freeze`. Jeśli konieczne jest użycie modyfikowalnych obiektów jako kluczy, po każdej dokonanej w nich zmianie należy wywołać metodę `rehash` z klasy Hash.

3.5. Zakresy

Obiekty klasy Range reprezentują wartości mieszczące się pomiędzy dwiema wartościami brzegowymi. Literał zakresowy składa się z dwóch wartości rozdzielonych dwiema lub trzema kropkami. Dwie kropki oznaczają, że zakres jest przedziałem zamkniętym, a więc wartość końcowa wchodzi w jego skład; trzy kropki — że zakres to przedział prawostronnie otwarty, a więc wartość końcowa nie wchodzi w jego skład:

```
1..10      # Liczby całkowite od 1 do 10, wliczając 10.  
1.0...10.0 # Liczby pomiędzy 1.0, a 10.0, wyłączając 10.0.
```

Do sprawdzenia, czy dana liczba należy do określonego przedziału, służy metoda `include?` (poniżej opisane zostały jej alternatywy):

```
cold_war = 1945..1989  
cold_war.include? birthdate.year
```

Z definicją zakresu związane jest pojęcie porządku. Jeśli zakres reprezentuje wartości między dwoma punktami brzegowymi, musi istnieć jakiś sposób na porównywanie ich z wartościami brzegowymi. W języku Ruby służy do tego operator `<=>` porównujący swoje operandy i zwracający wartość -1, 0 lub 1 w zależności od ich wzajemnego ułożenia (lub równości). Klasy takie jak liczbowe czy łańcuchowe, które są uporządkowane, definiują operator `<=>`. Wartość może zostać użyta jako brzeg zakresu tylko wówczas, gdy działa z tym operatorem. Punkty brzegowe zakresu i wartości w zakresie z reguły są tego samego typu. Jednak z technicznego punktu widzenia każda wartość zgodna z operatorami wartości brzegowych `<=>` może wchodzić w skład zakresu.

Głównym celem zakresów jest porównywanie, czy dana wartość znajduje się w przedziale, czy poza nim. Drugim ważnym zastosowaniem jest iteracja. Jeśli klasa punktów brzegowych zakresu definiuje metodę `succ` (od ang. *successor*), istnieje dyskretny zbiór elementów zakresu, po których można iterować za pomocą metod `each`, `step` oraz metod z modułu `Enumerable`. Weź na przykład zakres 'a'..'c':

```
r = 'a'..'c'  
r.each { |l| print "[#{l}]"}      # Drukuję "[a][b][c]".  
r.step(2) { |l| print "[#{l}]"}    # Drukuję "[a][c]".  
r.to_a                          #=> ['a','b','c']: Moduł Enumerable udostępnia metodę to_a.
```

Kod ten działa, ponieważ klasa String definiuje metodę `succ`. Wynikiem operacji `'a'.succ` jest `'b'`, a `'b'.succ` jest `'c'`. Zakresy, po których można iterować w ten sposób, to **zakresy dyskretne**. Zakresy, których punkty brzegowe nie definiują metody `succ`, nie umożliwiają iteracji, a więc można je nazwać zakresami **ciągłymi**. Zakresy, których punkty brzegowe to liczby całkowite, są dyskretne, natomiast zakresy z liczbami zmienoprzecinkowymi na brzegach są ciągłe.

W typowych programach Ruby najczęściej używane są zakresy z liczbami całkowitymi na brzegach. Ponieważ są dyskretne, zakresy całkowitoliczbowe mogą być używane do indeksowania łańcuchów i tablic. Są też wygodnym sposobem na reprezentację przeliczalnych (`Enumerable`) kolekcji rosnących wartości.

Warto zauważyć, że literał zakresowy został przypisany do zmiennej, a następnie poprzez tę zmienną zostały na jego rzecz wywołane metody. Aby wywołać metodę bezpośrednio na rzecz zakresu, należy jego literał umieścić w nawiasach. W przeciwnym wypadku metoda zostanie wywołana na rzecz punktu końcowego tego zakresu, zamiast na rzecz całego obiektu typu Range:

```
1..3.to_a    # Wywołuje metodę to_a na rzecz liczby 3.  
(1..3).to_a  # => [1,2,3].
```

3.5.1. Sprawdzanie przynależności do zakresu

Klasa Range udostępnia metody pozwalające sprawdzić, czy dowolna wartość należy do zbioru. Istotne jest, że przynależność do zakresu można zdefiniować na dwa różne sposoby, co ma związek z różnicą pomiędzy zakresem dyskretnym a zakresem ciągłym. Wartość x należy do zakresu begin..end, jeśli:

```
begin <= x <= end
```

x należy do zakresu begin...end (z trzema kropkami), jeżeli:

```
begin <= x < end
```

Obie wartości brzegowe zakresu muszą implementować operator `<=>`. Zatem niniejsza definicja dotyczy wszystkich obiektów typu Range i nie wymaga, aby punkty brzegowe implementowały metodę succ. Nazwiemy ją testem przynależności do zakresu ciągłego.

Druga definicja przynależności (przynależność do zbioru dyskretnego) jest uzależniona od metody succ. Traktuje ona obiekt Range begin..end jako zbiór wartości begin, begin.succ, begin.succ.succ itd. Zgodnie z tą definicją przynależność do zakresu jest równoważna z przynależnością do zbioru. Wartość x należy do zbioru tylko wówczas, gdy jest jedną z wartości zwróconych przez jedno z wywołań metody succ. Warto wiedzieć, że sprawdzanie przynależności do zakresu dyskretnego jest potencjalnie znacznie bardziej czasochłonne niż sprawdzanie przynależności do zakresu ciągłego.

Po zapoznaniu się z powyższymi informacjami możemy przejść do opisu metod klasy Range sprawdzających przynależność. Ruby 1.8 obsługuje dwie metody `include?` i `member?`. Są to synonimy sprawdzające przynależność do zakresu ciągłego:

```
r = 0...100      # Zakres liczb całkowitych od 0 do 99.  
r.member? 50     # => true: 50 należy do tego zakresu.  
r.include? 100   # => false: 100 nie mieści się w tym zakresie.  
r.include? 99.9  # => true: 99.9 to mniej niż 100.
```

W Ruby 1.9 sytuacja jest inna. W tej wersji języka wprowadzono nową metodę o nazwie `cover?`, która działa tak jak `include?` i `member?` w Ruby 1.8 — zawsze przeprowadza test przynależności do zakresu ciągłego. Metody `include?` i `member?` są nadal synonimami w Ruby 1.9. Jeśli punkty brzegowe zakresu są liczbami, metody te przeprowadzają test przynależności do zakresu ciągłego, tak jak w Ruby 1.8. Jednak gdy wartości brzegowe nie są liczbami, metody te stosują test przynależności do zakresu dyskretnego. Zmiany te zostały zilustrowane za pomocą dyskretnego zakresu łańcuchów (aby sprawdzić, jak działa metoda succ, można użyć narzędzia ri):

```
triples = "AAA".."ZZZ"  
triples.include? "ABC"          # true; szybka w 1.8, ale wolna w 1.9.  
triples.include? "ABCD"         # true w 1.8, false w 1.9.  
triples.cover? "ABCD"          # true i szybka w 1.9.  
triples.to_a.include? "ABCD"    # false i wolna w 1.8 i 1.9.
```

W praktyce większość zakresów posiada liczbowe punkty brzegowe, a zmiany w API klasy Range między wersjami Ruby 1.8 a 1.9 mają niewielkie znaczenie.

3.6. Symbole

Typowa implementacja Ruby posiada tablicę symboli, w której przechowywane są nazwy wszystkich znanych językowi klas, metod i zmiennych. Dzięki temu interpreter może uniknąć większości operacji porównywania łańcuchów — odwołuje się do nazw metod, na przykład poprzez ich położenie w tablicy symboli. W ten sposób względnie czasochłonna operacja na łańcuchach zostaje zamieniona na stosunkowo efektywną operację na liczbach całkowitych.

Symbole nie są wyłączną wewnętrzną własnością interpretera. Mogą być także używane w programach Ruby. Obiekt klasy `Symbol` wskazuje na symbol. Literał symbolu jest zbudowany z dwukropka poprzedzającego identyfikator:

```
:symbol                      # Literal symbolu.  
:"symbol"                    # Ten sam literal.  
:'another long symbol'      # Cudzysłów pozwalają na tworzenie symboli ze spacjami.  
s = "string"                 # Symbol :string.  
sym = :"#{s}"                # Symbol :string.
```

Literaly symboli obsługują także składnię `%s`, która pozwala na zastosowanie dowolnych ograniczników w podobny sposób jak `%q` i `%Q` w przypadku literałów łańcuchowych:

```
%s[""]          # Tak samo jak :"".
```

Symbole są często stosowane w odwołaniach do nazw metod w kodzie refleksyjnym. Założymy na przykład, że chcesz dowiedzieć się, czy jakiś obiekt implementuje metodę `each`:

```
o.respond_to? :each
```

Oto jeszcze jeden przykład. Poniższa procedura sprawdza, czy dany obiekt obsługuje określoną metodę. Jeśli tak, wywołuje ją:

```
name = :size  
if o.respond_to? name  
  o.send(name)  
end
```

Obiekt klasy `String` można przekonwertować na obiekt klasy `Symbol`, używając metody `intern` lub `to_sym`. Do wykonania operacji odwrotnej służy metoda o nazwie `to_s`, która posiada alias o nazwie `id2name`:

```
str = "string"    # Na początku jest łańcuch.  
sym = str.intern  # Konwersja na symbol.  
sym = str.to_sym  # Inny sposób na zrobienie tego samego.  
str = sym.to_s    # Konwersja z powrotem na łańcuch.  
str = sym.id2name # Inny sposób na zrobienie tego samego.
```

Dwa łańcuchy mogą mieć taką samą treść, a być dwoma całkiem odrębnymi obiektami. W przypadku symboli taka sytuacja nie może wystąpić. Dwa łańcuchy o takiej samej treści zostaną przekonwertowane na dokładnie ten sam obiekt klasy `Symbol`. Dwa różne obiekty klasy `Symbol` zawsze mają różną treść.

Pisząc kod, w którym łańcuchy są wykorzystywane nie ze względu na ich treść, ale jako rodzaj unikatowego identyfikatora, należy rozważyć użycie zamiast nich symboli. Na przykład zamiast pisać metodę, której argumentem musi być łańcuch "AM" lub "PM", można utworzyć metodę, której argumentem musi być symbol `:AM` lub `:PM`. Porównywanie dwóch obiektów

klasy `Symbol` jest znacznie szybsze niż porównywanie łańcuchów. Dlatego generalnie jako klucze tablic asocjacyjnych preferowane są symbole, a nie łańcuchy. W Ruby 1.9 klasa `Symbol` definiuje kilka metod klasy `String`, jak `length` i `size`, operatory porównywania, a nawet operatory `[]` i `=~`. Dzięki temu symbole często można stosować zamiennie z łańcuchami i mogą one być używane jako rodzaj niemodyfikowalnych (i nieusuwanych przez system usuwania nieużytków) łańcuchów.

3.7. Słowa kluczowe `true`, `false` i `nil`

Wiadomo już z rozdziału 2.1.5, że `true`, `false` i `nil` to słowa kluczowe. `true` i `false` to wartości logiczne reprezentujące prawdę i fałsz, tak lub nie lub włączenie i wyłączenie czegoś. `nil` to specjalna wartość, która oznacza brak wartości.

Wartością każdego z tych słów kluczowych jest specjalny obiekt. `true` jest obiektem będącym egzemplarzem singletonowej klasy `TrueClass`. Podobnie `false` i `nil` są singletonowymi egzemplarzami klas `FalseClass` i `NilClass`. Należy zauważyć, że w języku Ruby nie ma klasy `Boolean`. Nadklassą klas `TrueClass` i `FalseClass` jest klasa `Object`.

Aby sprawdzić, czy dana wartość to `nil`, wystarczy porównać ją z `nil` lub użyć metody `nil?`:

```
o == nil    # Czy o ma wartość nil?  
o.nil?     # Inny sposób sprawdzenia.
```

Należy zauważyć, że słowa kluczowe `true`, `false` i `nil` odnoszą się do obiektów, nie liczb. `false` i `nil` nie są równoważne z zerem, a `true` nie jest równoważne z jedynką. Kiedy w języku Ruby oczekiwana jest wartość logiczna, `nil` jest równoważne ze słowem kluczowym `false`, a każde słowo inne niż `nil` lub `false` działa jak `true`.

3.8. Obiekty

Ruby jest językiem czysto obiektowym — wszystkie wartości w nim zawarte są obiektami i nie ma rozróżnienia między typami podstawowymi a typami obiektowymi tak jak w wielu innych językach. W Ruby wszystkie obiekty dziedziczą po klasie o nazwie `Object` i implementują metody zdefiniowane w tej klasie. Niniejszy podrozdział opisuje wspólne własności wszystkich obiektów w języku Ruby. W niektórych miejscach zagęszczenie informacji jest bardzo duże, ale jest to lektura obowiązkowa. Informacje zawarte w tym podrozdziale są fundamentalne.

3.8.1. Referencje do obiektów

Wykonując działania związane z obiektami w języku Ruby, w rzeczywistości działa się na **referencjach** do obiektów. Obróbce nie jest poddawany sam obiekt, a jego referencja⁴. Przypisanie wartości do zmiennej nie powoduje skopiowania do niej obiektu. Zapisywana jest w niej tylko referencja (odwołanie) do obiektu. Wyjaśnimy to na przykładzie:

⁴ Osoby znające język C lub C++ mogą o referencjach myśleć jak o wskaźnikach — adresach obiektów w pamięci. Należy jednak pamiętać, że w Ruby nie używa się wskaźników. W tym języku referencje są nieprzezroczyste i stanowią wewnętrzną własność implementacji. Nie ma możliwości sprawdzenia adresu wartości, wyłuskania wartości lub wykonywania działań arytmetycznych na wskaźnikach.

```
s = "Ruby" # Utworzenie obiektu klasy String. Zapisanie referencji do niego w zmiennej s.  
t = s # Skopiowanie referencji do zmiennej t. s i t wskazują ten sam obiekt.  
t[-1] = "" # Modyfikacja obiektu poprzez referencję.  
print s # Dostęp do zmodyfikowanego obiektu przez referencję s. Drukuje "Rub".  
t = "Java" # t wskazuje teraz inny obiekt.  
print s, t # Drukuje "RubJava".
```

Kiedy do metody jest przekazywany obiekt, w rzeczywistości przekazywana jest jego referencja. Nie jest to sam obiekt ani referencja do referencji do obiektu. Innymi słowy, argumenty są przekazywane do metod **przez wartość**, a nie **przez referencję**, ale te przekazywane wartości są referencjami do obiektów.

Ponieważ do metod przekazywane są referencje, metody mogą za ich pomocą modyfikować wskazywane przez nie obiekty. Rezultat tych modyfikacji jest widoczny po zwróceniu przez metodę wartości.

3.8.1.1. Wartości bezpośrednie

Napisaliśmy, że wszystkie wartości w języku Ruby są obiektami, a wszystkie obiekty są przetwarzane za pośrednictwem referencji. Jednak w implementacji referencji obiekty klas Fixnum i Symbol są w rzeczywistości wartościami bezpośredniimi. Żadna z tych klas nie posiada metod mutacyjnych, dlatego obiekty tych klas nie mogą być modyfikowane. W związku z tym nie ma sposobu na określenie, czy są one przetwarzane przez wartość, czy przez referencję.

Istnienie wartości bezpośrednich należy traktować jako szczegół implementacyjny. Jedyna praktyczna różnica między wartościami bezpośredniimi a referencyjnymi jest taka, że te pierwsze nie mogą posiadać metod singletonowych (metody singletonowe zostały opisane w podrozdziale 6.1.4).

3.8.2. Czas istnienia obiektu

Wbudowane klasy Ruby opisane w niniejszym rozdziale posiadają składnie literałowe, a ich egzemplarze są tworzone poprzez wpisanie ich wartości w kodzie. Obiekty innych klas muszą być tworzone jawnie. Najczęściej używa się w tym celu metody new:

```
myObject = myClass.new
```

new jest metodą klasy Class. Przydziela pamięć dla nowego obiektu, a następnie inicjuje jego stan za pomocą wywołania metody initialize. Argumenty metody new są bezpośrednio przekazywane do metody initialize. Większość klas udostępnia initialize wykonującą wszelkie czynności inicjacyjne niezbędne przy tworzeniu ich egzemplarzy.

Metody new i initialize są domyślnym sposobem tworzenia nowych obiektów, ale klasa może udostępniać także inne metody zwracające jej egzemplarze nazywane metodami fabrykującymi (fabrykami). Więcej informacji na temat metod new, initialize i fabryk znajduje się w podrozdziale 7.4.

Obiektów w języku Ruby nie trzeba usuwać jawnie jak w językach C i C++. Ruby wykorzystuje specjalny system usuwający nieużywane obiekty do automatycznego niszczenia obiektów, które nie są już potrzebne. Obiekt staje się kandydatem do zniszczenia, kiedy jest nieosiągalny, to znaczy gdy w żadnym osiągalnym obiekcie nie ma do niego referencji.

Dzięki zastosowaniu systemu usuwania nieużywanych obiektów (ang. *garbage collector*) programy w języku Ruby są mniej podatne na błędy wycieku pamięci niż programy napisane w językach wymagających jawnego niszczenia obiektów i zwalniania pamięci. System usuwania nieużytków nie gwarantuje jednak, że wycieki nigdy nie wystąpią — każdy fragment kodu tworzący długotrwałe referencje do obiektów, które w przeciwnym razie byłyby krótkotrwale, może stanowić potencjalne źródło wycieku pamięci. Jako przykład niech posłuży tablica asocjacyjna użyta do implementacji pamięci podręcznej. Jeśli pamiętać ta nie jest czyszczona za pomocą jakiegoś algorytmu usuwającego najstarsze elementy, to obiekty w niej zapisane są dostępne tak długo, jak długo dostępna jest sama pamięć. W sytuacji, gdy tablica jest dostępna za pośrednictwem zmiennej globalnej, będzie ona osiągalna dotąd, dopóki będzie działał interpreter Ruby.

3.8.3. Tożsamość obiektów

Każdy obiekt posiada identyfikator — liczbę typu Fixnum — który można sprawdzić za pomocą metody `object_id`. Wartość zwrócona przez tę metodę jest stałą unikatową przez cały cykl życia obiektu. Dopóki obiekt jest osiągalny, cały czas ma ten sam identyfikator i żaden inny obiekt nie będzie mógł mieć takiego samego identyfikatora.

Metoda `id` jest wycofywanym synonimem metody `object_id`. Ruby 1.8 zgłasza ostrzeżenie przy jej użyciu. W Ruby 1.9 nie ma jej w ogóle.

Metoda `_id_` jest prawidłowym synonimem metody `object_id`. Służy jako ratunek na wypadek, gdyby metoda `object_id` nie została zdefiniowana lub przesłonięta.

Implementacja metody `hash` w klasie `Object` zwraca identyfikator obiektu.

3.8.4. Klasa i typ obiektu

W języku Ruby istnieje kilka sposobów na sprawdzenie klasy obiektu. Najprostszy polega na zapytaniu wprost:

```
o = "test"    # To jest wartość.  
o.class      # Zwraca obiekt reprezentujący klasę String.
```

Można także sprawdzić, jaka klasa jest nadklassą klasy dowolnego obiektu:

```
o.class          # String: o jest obiektem klasy String.  
o.class.superclass  # Object: nadklassą klasy String jest klasa Object.  
o.class.superclass.superclass # nil: klasa Object nie ma nadklaś.
```

W Ruby 1.9 klasa `Object` nie jest już klasą najwyższego poziomu:

```
# Ruby 1.9 only.  
Object.superclass        # BasicObject: klasa Object ma nadklaś w Ruby 1.9.  
BasicObject.superclass   # nil: klasa BasicObject nie ma nadklaś.
```

Więcej informacji na temat klasy `BasicObject` znajduje się w podrozdziale 7.3.

Najprostszym sposobem na sprawdzenie klasy obiektu jest bezpośrednie porównanie:

```
o.class == String      # true, jeśli o należy do klasy String.
```

Metoda `instance_of?` robi dokładnie to samo, a przy tym jest nieco bardziej elegancka:

```
o.instance_of? String  # true, jeśli o należy do klasy String.
```

Z reguły przy sprawdzaniu klasy obiektu potrzebne są też informacje, czy jest on egzemplarzem jakiejś podklasy tej klasy. Aby to sprawdzić, należy użyć metody `is_a?` lub jej synonimu `kind_of?:`

```
x = 1          # Wartość, która będzie przetwarzana.  
x.instance_of? Fixnum # true: egzemplarz klasy Fixnum.  
x.instance_of? Numeric # false: metoda instance_of? nie sprawdza w głębi hierarchii.  
x.is_a? Fixnum # true: x należy do klasy Fixnum.  
x.is_a? Integer # true: x należy do klasy Integer.  
x.is_a? Numeric # true: x należy do klasy Numeric.  
x.is_a? Comparable # true: działa także z modułami domieszkowymi.  
x.is_a? Object # true dla każdej wartości x.
```

Klasa `Class` definiuje operator `==` w taki sposób, że można go używać zamiast metody `is_a?:`

```
Numeric === x # true: x jest Numeric.
```

Ta metoda jest unikatową właściwością języka Ruby i jest raczej mniej czytelna niż tradycyjna metoda `is_a?:`.

Każdy obiekt w Ruby należy do jakiejś ścisłe określonej klasy, która pozostaje niezmieniona przez cały cykl życia obiektu. Natomiast **typ** obiektu jest bardziej płynny. Jest on związany z jego klasą, ale klasa to tylko część informacji o typie. Mówiąc typ obiektu, mamy na myśli zestaw działań charakteryzujących ten obiekt. Innymi słowy, typ obiektu to zestaw metod, które można wywoływać na jego rzecz (definicja ta jest nieprecyzyjna, ponieważ znaczenie mają nie tyle nazwy metod, co typy argumentów, które te metody mogą przyjmować).

Dla programisty klasa obiektu często nie ma znaczenia. Ważne jest, czy można na jego rzecz wywołać określone metody. Weźmy na przykład operator `<<`. Tablice, łańcuchy, pliki i inne związane z wejściem i wyjściem klasy definiują go jako operator dołączający. Pisząc metodę wysyłającą tekst, można napisać ją rodzajowo, aby używała tego operatora. Wtedy można ją wywoływać z dowolnym argumentem implementującym operator `<<`. Nieważna jest klasa tego argumentu, tylko to, czy można do niego coś dołączyć. Można to sprawdzić, używając metody `respond_to?:`:

```
o.respond_to? :"<<" # true, jeśli o posiada operator <<.
```

Wadą tego rozwiązania jest to, że sprawdzana jest tylko nazwa metody, a nie jej argumenty. Na przykład w klasach `Fixnum` i `Bignum` operator `<<` jest operatorem przesunięcia w lewo, a jego argumentem powinna być liczba, a nie łańcuch. Przy wywoaniu metody `respond_to?` wydaje się, że do obiektów całkowitoliczbowych można coś dodać, ale próba dodania do nich łańcucha kończy się błędem. Nie ma uniwersalnego rozwiązania tego problemu, ale w tym przypadku można poradzić sobie, eliminując obiekty klasy `Numeric` za pomocą metody `is_a?:`

```
o.respond_to? :"<<" and not o.is_a? Numeric
```

Innym przykładem różnicy między klasą a typem jest klasa `StringIO` (z biblioteki standardej Ruby). Umożliwia ona odczytywanie i zapis obiektów łańcuchowych w taki sposób, jakby były obiektami klasy `IO`. Klasa `StringIO` naśladuje API `IO` — obiekty `StringIO` definiują takie same metody jak obiekty `IO`. Jednak klasa `StringIO` nie jest podklassą klasy `IO`. Jeśli napiszesz metodę przyjmującą jako argument strumień i sprawdzisz klasę tego argumentu za pomocą testu `is_a? IO`, Twoja metoda nie będzie działała z argumentami klasy `StringIO`.

Koncentrowanie się na typach zamiast klasach prowadzi do stylu programowania w Ruby nazywanego kaczym typowaniem (ang. *duck typing*). Przykłady takiego typowania przedstawione są w rozdziale 7.

3.8.5. Porównywanie obiektów

W języku Ruby jest zaskakująco dużo sposobów na porównywanie obiektów. Ważne jest, aby je wszystkie znać i wiedzieć, kiedy użyć której metody.

3.8.5.1. Metoda equal?

Metoda `equal?` jest zdefiniowana w klasie `Object`. Sprawdza, czy dwie wartości odwołują się do dokładnie tego samego obiektu. Dla dowolnych dwóch różnych obiektów metoda ta zawsze zwraca wartość `false`:

```
a = "Ruby"      # Jedna referencja do jednego obiektu klasy String.  
b = c = "Ruby"  # Dwie referencje do innego obiektu klasy String.  
a.equal?(b)     # false: a i b są różnymi obiektami.  
b.equal?(c)     # true: b i c odwołują się do tego samego obiektu.
```

Zgodnie z konwencją metoda `equal?` nigdy nie jest przesłaniana w podklasach.

Innym sposobem na sprawdzenie, czy dwa obiekty są w rzeczywistości tym samym obiektem, jest przeanalizowanie ich identyfikatorów za pomocą metody `object_id`:

```
a.object_id == b.object_id  # Działa jak wywołanie a.equal?(b).
```

3.8.5.2. Operator ==

Operator `==` jest najczęściej używanym sposobem na porównywanie obiektów. W klasie `Object` jest on synonimem metody `equal?`. Sprawdza, czy dwie referencje do obiektu są identyczne. Większość klas przeddefiniuje ten operator, aby umożliwić porównywanie odrębnych egzemplarzy:

```
a = "Ruby"      # Obiekt klasy String.  
b = "Ruby"      # Inny obiekt klasy String z taką samą treścią.  
a.equal?(b)     # false: a i b nie odwołują się do tego samego obiektu.  
a == b         # true: ale oba te odrębne obiekty mają identyczne wartości.
```

Należy zauważyć, że pojedynczy znak równości w powyższym kodzie służy jako operator przypisania. W Ruby do porównywania obiektów używa się dwóch znaków równości (podobna konwencja jest stosowana w wielu innych językach programowania).

Większość standardowych klas Ruby definiuje operator `==` w taki sposób, aby implementował rozsądную definicję równości. Zaliczają się do nich również klasy `Array` i `Hash`. Według operatora `==` dwie tablice są równe, jeśli mają po tyle samo elementów i wszystkie odpowiadające sobie elementy są sobie równe. Dwie tablice asocjacyjne są równe, gdy zawierają takie same liczby par klucz-wartość, a klucze i wartości również są sobie równe (wartości są porównywane przez operator `==`, natomiast klucze tablic asocjacyjnych przez metodę `eql?` opisaną w dalszej części niniejszego rozdziału).

Równość dla programistów Javy

Programiści Javy są przyzwyczajeni do używania operatora `==` do sprawdzania, czy dwa obiekty są tym samym obiektem, oraz metody `equals` do przeanalizowania, czy dwa odrębne obiekty mają taką samą wartość. W języku Ruby jest na odwrót.

Klasy Numeric dokonują prostej konwersji typu w swoich operatorach ==, dzięki czemu na przykład obiekty Fixnum 1 i Float 1.0 są określane jako równe. Normalnie operator == w takich klasach jak String czy Array wymaga, aby oba jego operandy należały do tej samej klasy. Jeśli prawy operand definiuje funkcję konwertującą to_str lub to_ary (zobacz podrozdział 3.8.7), to operatory te wywołują operator == zdefiniowany przez prawy operand i pozwalają obiektowi zdecydować, czy jest równy znajdującemu się po lewej stronie łańcuchowi lub tablicy. Dzięki temu możliwe jest (aczkolwiek nie jest to zbyt często praktykowane) zdefiniowanie klasy z operacją porównywania łańcuchowego lub tablicowego.

Operator != (różny) służy w Ruby do sprawdzania, czy obiekty nie są równe. Kiedy Ruby natyka operator !=, używa operatora ==, a następnie odwraca wynik. Oznacza to, że w klasie musi być zdefiniowany tylko operator ==. Operator != jest w prezencie od Ruby. W Ruby 1.9 można jednak jawnie zdefiniować własny operator !=.

3.8.5.3. Metoda eql?

Metoda eql? jest zdefiniowana w klasie Object jako synonim metody equal?. Klasy ją przesłaniające zazwyczaj używają jej jako ścisłej wersji operatora ==, która nie dokonuje konwersji. Na przykład:

```
1 == 1.0      # true: obiekty Fixnum i Float mogą być równe,  
1.eql?(1.0) # false: ale nie według metody eql?!
```

Klasa Hash sprawdza za pomocą metody eql?, czy dwa klucze są równe. Jeśli dwa obiekty są równe według metody eql?, ich metody hash muszą zwracać tę samą wartość. Zazwyczaj pisząc klasę i definiując operator ==, można napisać metodę hash, a metodę eql? zdefiniować w taki sposób, aby korzystała z operatora ==.

3.8.5.4. Operator ===

Operator === jest nazywany operatorem instrukcji case (ang. *case equality*) i jest używany do sprawdzania, czy docelowa wartość instrukcji case pasuje do którejś z klauzul when tej instrukcji (instrukcja case jest instrukcją rozwidlającą się na wiele kierunków i została opisana w rozdziale 5.).

Klasa Object definiuje domyślny operator === w taki sposób, że wywołuje on operator ==. Dlatego w wielu klasach równość szczególna jest tym samym co równość ==. Niektóre jednak kluczowe klasy definiują operator === inaczej. W tych przypadkach jest to raczej operator przynależności lub dopasowania. W klasie Range operator === sprawdza, czy wartość należy do określonego zakresu; w klasie Regexp — czy dany łańcuch pasuje do wyrażenia regularnego; natomiast w klasie Class — czy określony obiekt jest egzemplarzem tej klasy. W Ruby 1.9 operator === z klasy Symbol zwraca wartość true, jeśli prawy operand jest tym samym symbolem co lewy lub jest łańcuchem przechowującym ten sam tekst. Przykłady:

```
(1..10) === 5      # true: 5 należy do zakresu 1..10.  
/\d+/ === "123"   # true: łańcuch pasuje do wyrażenia regularnego.  
String === "s"    # true: "s" jest egzemplarzem klasy String.  
:s === "s"        # true w Ruby 1.9.
```

Rzadko spotyka się jawnie użycie operatora === w takiej roli. Częściej jest on używany niejawnie w instrukcji case.

3.8.5.5. Operator =~

Operator =~ jest zdefiniowany w klasach `String` i `Regexp` (oraz `Symbol` w Ruby 1.9). Jego funkcja jest dopasowywanie wzorców i nie ma on w rzeczywistości nic wspólnego z operatorem porównywania. Jednak ponieważ w jego skład wchodzi znak równości, został on opisany tutaj, aby zachować porządek. Klasa `Object` definiuje pustą wersję operatora =~, która zawsze zwraca wartość `false`. Można zdefiniować go w swojej klasie, jeśli klasa ta definiuje na przykład jakiegoś rodzaju operację dopasowywania wzorców lub ma coś wspólnego z przybliżoną równością. Odwrotnością operatora =~ jest operator !~. Można go zdefiniować w Ruby 1.9, ale nie w 1.8.

3.8.6. Porządkowanie obiektów

Praktycznie każda klasa może zdefiniować metodę `==` porównującą jej obiekty. Niektóre klasy mogą także definiować porządek. To znaczy dowolne dwa egzemplarze takiej klasy muszą być równe lub jeden musi być mniejszy od drugiego. Najbardziej oczywistymi klasami, dla których jest definiowany taki porządek, są klasy liczbowe. Łańcuchy także są uporządkowane według kodów składających się na nie znaków (w przypadku znaków z zestawu ASCII jest to rodzaj porządku alfabetycznego z rozróżnianiem małych i wielkich liter). Jeśli klasa definiuje porządek, jej obiekty mogą być porównywane i sortowane.

W języku Ruby porządek w klasach jest definiowany przy użyciu operatora `<=>`. Powinien on zwracać wartość `-1` — jeśli jego lewy operand jest mniejszy niż prawy, `0` — gdy operandy są równe lub `1` — jeżeli lewy operand jest większy niż prawy. Jeśli podanych operandów nie można porównać w żaden sensowny sposób (na przykład gdy prawy operand należy do innej klasy), operator ten powinien zwrócić wartość `nil`:

```
1 <=> 5      # -1.  
5 <=> 5      # 0.  
9 <=> 5      # 1.  
"1" <=> 5    # nil: nie można porównywać liczb całkowitych iłańcuchów.
```

Operator `<=>` w zupełności wystarcza do porównywania wartości, ale nie jest zbyt intuicyjny. Dlatego klasy go definiujące zazwyczaj zawierają także moduł `Comparable` jako domieszkę (moduły i domieszki zostały opisane w podrozdziale 7.5.2). Domieszka `Comparable` definiuje następujące operatory w kategoriach `<=>`:

<	operator mniejszości
<=	mniejszy lub równy
==	równy
>=	większy lub równy
>	operator większości

Moduł `Comparable` nie zawiera definicji operatora `!=`, który Ruby automatycznie definiuje jako negację operatora `==`. Poza tymi operatorami porównywania moduł `Comparable` zawiera jeszcze przydatną metodę porównującą o nazwie `between?`:

```
1.between?(0,10)  # true: 0 <= 1 <= 10.
```

Jeśli operator `<=>` zwraca wartość `nil`, wszystkie utworzone na jego podstawie operatory zwracają wartość `false`. Przykładem jest specjalna wartość klasy `Float` `NaN` (Not-a-Number — nie liczba):

```
nan = 0.0/0.0;      # Wynik dzielenia zera przez zero nie jest liczbą.  
nan < 0            # false: to nie jest mniejsze niż zero.  
nan > 0            # false: to nie jest większe niż zero.
```

```
nan == 0          #false: to nie jest tyle samo co zero.  
nan == nan       #false: to nie jest równe nawet samo sobie!  
nan.equal?(nan)  # To oczywiście jest prawda.
```

Należy zauważyc, że zdefiniowanie operatora `<=>` i dołączenie modułu `Comparable` powoduje zdefiniowanie w klasie operatora `==`. Niektóre klasy definiują własny operator `==` zazwyczaj wówczas, gdy jego implementacja może być bardziej efektywna niż działanie operatora `<=>`. Istnieje możliwość zdefiniowania klasy implementującej różne rodzaje równości w jej operatorach `==` i `<=>`. Klasa może na przykład za pomocą operatora `==` wykonywać porównywanie z rozróżnianiem wielkich i małych liter, a przy użyciu operatora `<=>` wykonywać bardziej naturalne porównywanie bez rozróżniania wielkich i małych liter. Najlepiej jednak, jeśli operator `<=>` zwraca wartość 0 wtedy i tylko wtedy, gdy `==` zwraca wartość `true`.

3.8.7. Konwersja obiektów

Wiele klas Ruby udostępnia metody zwracające reprezentacje obiektów jako wartości innych klas. Do najczęściej implementowanych i najszerzej znanych tego typu metod należy `to_s`, która konwertuje obiekt na łańcuch. Kolejne podrozdziały opisują różne rodzaje konwersji.

3.8.7.1. Konwersja jawną

Metody konwersji jawnej są używane w kodzie aplikacji do zamiany wartości jednej klasy na wartość innej klasy. Do najczęściej używanych metod w tej kategorii należą: `to_s`, `to_i`, `to_f` i `to_a`, które konwertują obiekty na klasy odpowiednio: `String`, `Integer`, `Float` i `Array`.

Standardowe metody z reguły nie wywołują tych metod automatycznie. Jeśli do wywołania metody przyjmującej jako argument obiekt klasy `String` zostanie przekazany obiekt innej klasy, metoda ta nie wywoła automatycznie metody `to_s`, aby dokonać konwersji (natomiast wartości interpolowane do łańcuchów w podwójnych cudzysłowach są automatycznie konwertowane za pomocą metody `to_s`).

Metoda `to_s` jest najważniejszą ze wszystkich metod konwertujących, ponieważ obiekty łańcuchowe są powszechnie stosowane w interfejsach użytkownika. Ważną alternatywą dla `to_s` jest metoda `inspect`. Ta pierwsza ma na celu zwracać reprezentacje obiektów nadających się do odczytu przez człowieka, zwłaszcza użytkownika końcowego. Natomiast `inspect` jest przeznaczona do celów debugowania, a więc powinna zwracać łańcuch pomocny dla programistów. Domyślna wersja metody `inspect` w klasie `Object` wywołuje metodę `to_s`.

W wersji 1.9 języka Ruby dostępne są także metody `to_c` oraz `to_r` konwertujące obiekt odpowiednio na obiekty klasy `Complex` oraz `Rational`.

3.8.7.2. Konwersja niejawna

Niektóre klasy mają bardzo dużo cech niektórych innych klas. Na przykład klasa `Exception` reprezentuje błąd lub nieoczekiwana sytuację w programie. Jej obiekty zawierają stosowne komunikaty o błędach. W Ruby 1.8 obiekty klasy `Exception` nie tylko można konwertować na klasę `String`. Są to podobne do łańcuchów obiekty, które w wielu sytuacjach można traktować tak, jakby były łańcuchami⁵. Na przykład:

⁵ Odradzamy jednak takie działanie. W Ruby 1.9 niejawna konwersja klasy `Exception` na `String` jest zabroniona.

```
# Tylko Ruby 1.8.  
e = Exception.new("Nie wyjątek")  
msg = "Błąd: " + e # Konkatenacja łańcucha i wyjątku.
```

Obiektów klasy `Exception` można używać z operatorem konkatenacji, ponieważ są podobne do łańcuchów. Nie da się tego zrobić z większością innych klas w Ruby. Powodem, dla którego w Ruby 1.8 obiekty klasy `Exception` mogą zachowywać się jak obiekty klasy `String`, jest to, iż klasa `Exception` implementuje metodę konwersji niejawną o nazwie `to_str`, a operator `+` zdefiniowany w klasie `String` wywołuje tę metodę na rzecz swojego prawego operandu.

Inne metody konwersji niejawnnej to: `to_int` — konwertuje obiekty na podobieństwo liczb całkowitych, `to_ary` — obiekty podobne do tablic oraz `to_hash` — obiekty podobne do tablic asocacyjnych. Niestety, warunki, w których te metody są wywoływane, nie są dobrze opisane w dokumentacji. Ponadto metody te nie są zaimplementowane w wielu standardowych klasach.

Zwróciliśmy wcześniej uwagę, że operator `==` może wykonywać słabą konwersję typów podczas porównywania obiektów. Operatory `==` w klasach `String`, `Array` i `Hash` sprawdzają, czy ich prawy operand należy do tej samej klasy co lewy. Jeśli tak, obiekty zostają porównane. Jeżeli nie, następuje sprawdzenie, czy prawy operand udostępnia metodę `to_str`, `to_ary` lub `to_hash`. Operatory nie wywołują tej metody, ale jeśli ona istnieje, wywołują metodę `==` prawnego operandu i pozwalają jej zdecydować, czy operand ten jest równy lewemu operandowi.

W Ruby 1.9 standardowe klasy `String`, `Array`, `Hash`, `Regexp` i `IO` udostępniają metodę klasową o nazwie `try_convert`. Metoda ta konwertuje swój argument, jeśli ten definiuje odpowiednią metodę konwertującą, lub w przeciwnym przypadku zwraca wartość `nil`. Wywołanie `Array.try_convert(o)` zwraca wywołanie `o.to_ary`, jeśli obiekt `o` definiuje tę metodę. W przeciwnym wypadku zwraca wartość `nil`. Metody `try_convert` są wygodne podczas pisania metod pozwalających na niejawną konwersję swoich argumentów.

3.8.7.3. Funkcje konwertujące

Moduł `Kernel` definiuje cztery metody konwertujące, które działają tak jak globalne funkcje konwersji. Funkcje te mają takie same nazwy — `Array`, `Float`, `Integer` i `String` — jak klasy, na które konwertują. Ich wyjątkowość polega na tym, że mają nazwy zaczynające się od wielkich liter.

Funkcja `Array` próbuje konwertować swój argument na tablicę, wywołując metodę `to_ary`. Jeśli metoda nie została zdefiniowana lub zwraca wartość `nil`, następuje próba wywołania metody `to_a`. Jeżeli metoda `to_a` nie została zdefiniowana lub zwraca wartość `nil`, funkcja `Array` zwraca nową tablicę, której jedynym elementem jest argument tej funkcji.

Funkcja `Float` konwertuje argumenty klasy `Numeric` bezpośrednio na klasę `Float`. Dla każdej wartości innej klasy niż `Numeric` wywołuje metodę `to_f`.

Funkcja `Integer` konwertuje swój argument na klasę `Fixnum` lub `Bignum`. Jeśli argument jest wartością klasy `Numeric`, następuje bezpośrednią konwersja. Wartości zmienoprzecinkowe nie są zaokrąglane, usuwa się tylko ich część ułamkową. W sytuacji gdy argument jest łańcuchem, funkcja ta szuka informacji o podstawie systemu liczenia (wiodące zero oznacza liczbę ósemkową, `0x` szesnastkową, a `0b` binarną) i odpowiednio konwertuje ten łańcuch. W przeciwnieństwie do metody `String.to_i` funkcja ta nie pozwala na stosowanie na końcu znaków innych niż cyfry. W przypadku argumentów każdego innego typu funkcja `Integer` najpierw próbuje konwersji za pomocą metody `to_int`, a później `to_i`.

W końcu funkcja `String` konwertuje swój argument na łańcuch, wywołując na jego rzecz metodę `to_s`.

3.8.7.4. Koercja typów przez operatory arytmetyczne

Typy liczbowe udostępniają metodę o nazwie `coerce`. Jej celem jest konwersja argumentu na taki sam typ jak obiekt klasy liczbowej, na rzecz którego została wywołana, lub konwersja obu obiektów na bardziej ogólny zgodny typ. Metoda `coerce` zawsze zwraca tablicę dwóch liczb tego samego typu. Pierwszy element tej tablicy to przekonwertowana wartość argumentu metody `coerce`. Drugi element to wartość (w razie potrzeby przekonwertowana), na rzecz której metoda ta została wywołana:

```
1.1.coerce(1)      # [1.0, 1.1]: Koercja z klasy Fixnum na Float.  
require "rational" # Użycie liczb wymiernych.  
r = Rational(1,3)  # Jedna trzecia jako liczba wymierna.  
r.coerce(2)        # [Rational(2,1), Rational(1,3)]: Fixnum na Rational.
```

Metoda `coerce` jest wykorzystywana przez operatory arytmetyczne. Na przykład operator `+` w klasie `Fixnum` nie rozpoznaje liczb klasy `Rational` (wymierne) i jeśli jego prawy operand należy do klasy `Rational`, operator ten nie wie, jak go dodać. Metoda `coerce` rozwiązuje ten problem. Operatory numeryczne działają w taki sposób, że jeśli nie znają typu swojego prawnego operandu, wywołują na jego rzecz metodę `coerce`, lewy argument przekazując jako jej argument. Wracając do przykładu dodawania obiektu klasy `Fixnum` do obiektu klasy `Rational`, metoda `coerce` klasy `Rational` zwraca tablicę dwóch wartości klasy `Rational`. Teraz operator `+` zdefiniowany w klasie `Fixnum` może wywołać operator `+` na rzecz elementów tej tablicy.

3.8.7.5. Konwersja typów logicznych

Wartości logiczne zasługują na odrębny podrozdział w kontekście konwersji typów. Język Ruby bardzo surowo podchodzi do wartości logicznych — wartości `true` i `false` mają metody `to_s`, które zwracają łańcuchy "true" i "false". Na tym kończy się lista metod konwertujących. Nie ma metody `to_b`, która konwertowałaby inne wartości na typ logiczny.

W niektórych językach wartość `false` jest równoważna z wartością `0` lub może zostać na nią przekonwertowana (i na odwrotnie). W języku Ruby wartości `true` i `false` są odrębnymi obiektami i nie ma żadnych metod niejawnie konwertujących inne wartości na `true` lub `false`. To jednak dopiero połowa historii. Operatory logiczne oraz instrukcje warunkowe i pętle, które używają wyrażeń logicznych, mogą działać z wartościami innymi niż `true` i `false`. Zasada jest prosta: w wyrażeniach logicznych każda wartość inna niż `false` lub `nil` działa jak `true` (ale nie jest na nią konwertowana). Wartość `nil` jest natomiast odpowiednikiem wartości `false`.

Załóżmy, że chcesz sprawdzić, czy zmienna `x` ma wartość `nil`, czy nie. W niektórych językach konieczne jest napisanie wyrażenia porównującego, które zwraca wartość `true` lub `false`:

```
if x != nil    # Wyrażenie "x != nil" zwraca do instrukcji if wartość true lub false.  
  puts x      # Drukuje x, jeśli jest to wartość zdefiniowana.  
end
```

Niniejszy fragment kodu działa, ale częściej wykorzystuje się fakt, że wszystkie wartości inne niż `nil` i `false` odpowiadają wartości `true`:

```
if x          # Jeśli wartość x nie jest nil,  
  puts x      # zostanie wydrukowana.  
end
```

Nie można zapomnieć, że wartości takie jak 0, 0.0 i pusty łańcuch "" w języku Ruby odpowiadają wartości true, co jest zaskakujące na przykład dla użytkowników języków C czy JavaScript.

3.8.8. Kopiowanie obiektów

Klasa Object udostępnia dwie blisko spokrewnione metody służące do kopowania obiektów. Zarówno metoda clone, jak i dup zwracają płytka kopię obiektu, na rzecz którego zostały wywołane. Jeśli kopiowany obiekt zawiera wewnętrzny stan odwołujący się do innych obiektów, kopiowane są tylko referencje do innych obiektów, a nie same obiekty.

Gdy kopiowany obiekt udostępnia metodę initialize_copy, metody clone i dup tworzą w pamięci nowy pusty egzemplarz tej klasy i wywołują na jego rzecz metodę initialize_copy. Obiekt, który ma być skopiowany, jest przekazywany jako argument, a konstruktor kopiący może zainicjować kopowanie w dowolny sposób. Na przykład metoda initialize_copy może rekursively skopiować dane wewnętrzne obiektu, dzięki czemu powstanie obiekt będący czymś więcej niż płytka kopią oryginału.

Metody clone i dup można także przedefiniować, aby kopiowały w dowolny sposób.

Między metodami dup a clone w klasie Object są dwie ważne różnice. Po pierwsze, metoda clone kopiuje zarówno stan zamrożony, jak i zanieczyszczony (opisane nieco dalej) obiektu, podczas gdy metoda dup kopiuje tylko stan zanieczyszczony. Wywołanie dup na rzecz zamrożonego obiektu zwraca niezamrożoną kopię. Po drugie, metoda clone kopiuje wszystkie metody singletonowe obiektu, podczas gdy metoda dup tego nie robi.

3.8.9. Szeregowanie obiektów

Stan obiektu można zapisać, przekazując go do metody klasowej Marshal.dump⁶. Jeśli jako drugi argument zostanie przekazany obiekt strumienia wejściowego lub wyjściowego, metoda ta zapisuje stan tego obiektu (i rekursively wszystkich obiektów, do których ten obiekt zawiera referencje) do tego strumienia. W przeciwnym wypadku zwraca stan zakodowany jako łańcuch binarny.

Aby przywrócić tak zaszeregowany obiekt, należy przekazać łańcuch lub strumień wejścia lub wyjścia zawierający ten obiekt do metody Marshal.load.

Szeregowanie obiektów (ang. *marshaling*) jest bardzo prostym sposobem na zapisywanie ich stanu do późniejszego użytku, a metody te można wykorzystać do utworzenia automatycznego formatu plików dla programów Ruby. Należy jednak zauważyć, że format binarny używany przez metody Marshal.dump i Marshal.load jest zależny od wersji Ruby i nowsze wersje tego języka mogą nie być w stanie odczytać starszych obiektów tego rodzaju.

Innym zastosowaniem metod Marshal.dump i Marshal.load jest tworzenie głębokich kopii obiektów:

```
def deepcopy(o)
  Marshal.load(Marshal.dump(o))
end
```

⁶ Słowo *marshal* jest w języku angielskim czasami pisane z dwoma literami „l” na końcu — *marshall*, *marshalled* itd. Należy pamiętać, że w języku Ruby nazwę tej metody pisze się z jednym „l” na końcu.

Zwróć uwagę, że pliki i strumienie wejścia-wyjścia, jak również obiekty klas Method i Binding są zbyt dynamiczne, aby można było je szeregować. Nie byłoby niezawodnego sposobu na przywrócenie ich stanu.

Język YAML (ang. *YAML Ain't Markup Language*) jest często używaną alternatywą dla modułu Marshal, która zapisuje obiekty w formacie tekstowym możliwym do odczytu przez człowieka (jak również ładuje je z formatu tekstopowego). Znajduje się w bibliotece standardowej i jego użycie wymaga dodania wiersza require 'yaml'.

3.8.10. Zamrażanie obiektów

Każdy obiekt można zamrozić za pomocą metody `freeze`. Zamrożony obiekt nie może zostać zmodyfikowany — żaden z jego wewnętrznych stanów nie może się zmienić, a wywołanie któregośkolwiek z jego metod mutacyjnych kończy się niepowodzeniem:

```
s = "lód"          # Łańcuchy są obiektami modyfikowalnymi.  
s.freeze          # Zamrożenie łańcucha, aby był niemodyfikowalny.  
s.frozen?        # true: obiekt został zamrożony.  
s.upcase!        # TypeError: nie można zmodyfikować zamrożonego łańcucha.  
s[0] = "mi"       # TypeError: nie można zmodyfikować zamrożonego łańcucha.
```

Zamrożenie obiektu klasowego uniemożliwia dodawanie metod do jego klasy.

Do sprawdzania, czy obiekt jest zamrożony, służy metoda `frozen?`. Po zamrożeniu obiektu nie ma sposobu na jego rozmrożenie. Jeśli obiekt zostanie skopiowany za pomocą metody `clone`, kopia również będzie zamrożona. Natomiast kopia wykonana przy użyciu metody `dup` nie będzie zamrożona.

3.8.11. Zanieczyszczanie obiektów

Aplikacje sieciowe często muszą badać dane otrzymywane od niezaufanych użytkowników w celu uniknięcia ataków typu SQL injection lub innych tego rodzaju zagrożeń. Ruby rozwiązuje ten problem w prosty sposób — każdy obiekt można oznać jako zanieczyszczony za pomocą metody `taint`. Wszystkie obiekty pochodzące od zanieczyszczonego obiektu również są zanieczyszczone. Aby sprawdzić, czy obiekt jest zanieczyszczony, należy użyć metody `tainted?`:

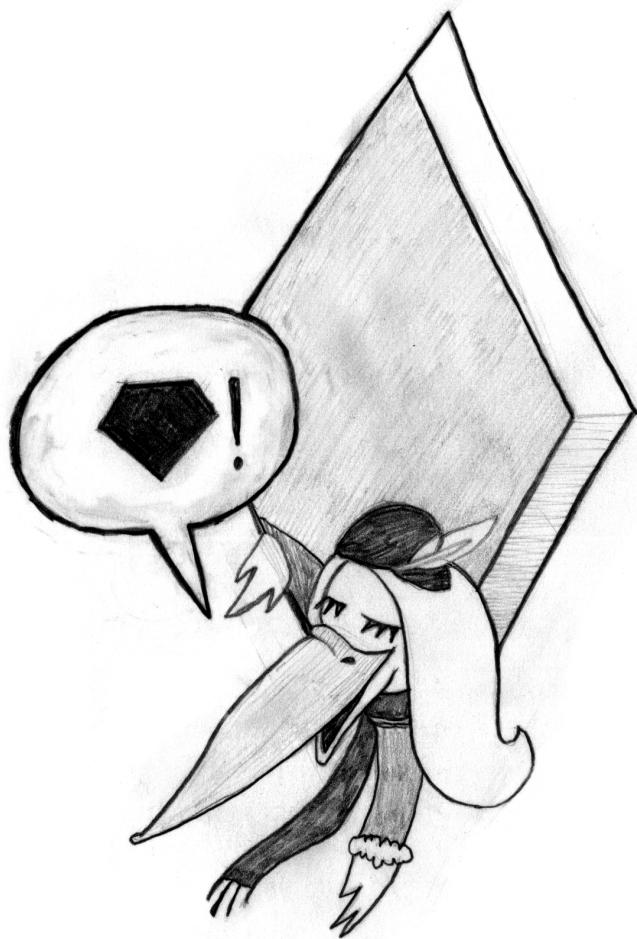
```
s = "niezaufany"  # Domyślnie obiekty nie są zanieczyszczone.  
s.taint           # Oznaczenie niezaufanego obiektu jako zanieczyszczony.  
s.tainted?       # true: obiekt jest zanieczyszczony.  
s.upcase.tainted? # true: obiekty pochodne są zanieczyszczone.  
s[3..4].tainted?  # true: podłańcuchy są zanieczyszczone.
```

Dane pochodzące od użytkownika — takie jak argumenty wiersza poleceń, zmienne środowiskowe oraz łańcuchy wczytywane przez metodę `gets` — są automatycznie oznaczane jako zanieczyszczone.

Kopie obiektów zanieczyszczonych robione za pomocą metod `clone` i `dup` pozostają zanieczyszczone. Obiekt zanieczyszczony można oczyścić, stosując metodę `untaint`. Oczywiście należy to robić wyłącznie wówczas, gdy po zbadaniu obiektu ma się pewność, że jest on bezpieczny.

Możliwości mechanizmu zanieczyszczania obiektów w języku Ruby można w pełni wykorzystać w połączeniu ze zmienną globalną `$SAFE`. Jeśli zmienna ta jest ustawiona na wartość większą od zera, Ruby nie zezwoli niektórym standardowym metodom na działanie z obiektami zanieczyszczonymi. Więcej szczegółów na temat zmiennej `$SAFE` znajduje się w rozdziale 10.

Wyrażenia i operatory



Wyrażenie to fragment kodu w języku Ruby, którego wartość jest obliczana przez interpreter. Poniżej znajduje się kilka przykładowych wyrażeń:

```
2          # Literal liczbowy.  
x          # Odwołanie do zmiennej lokalnej.  
Math.sqrt(2) # Wywołanie metody.  
x = Math.sqrt(2) # Przypisanie.  
x*x        # Mnożenie za pomocą operatora *
```

Jak widać, wyrażenia pierwotne — literały, odwołania do zmiennych i wywołania metod — można łączyć w większe wyrażenia za pomocą **operatorów**, takich jak operator przypisania czy mnożenia.

W wielu językach programowania rozróżnia się wyrażenia niskiego poziomu i wysokiego poziomu (instrukcje), takie jak instrukcje warunkowe i pętle. W tych językach instrukcje kontrolują przepływ sterowania w programie, ale nie mają wartości. Nie są obliczane, a wykonywane. W języku Ruby różnica pomiędzy instrukcjami a wyrażeniami nie jest jasna. W języku tym można wszystko potraktować jako wyrażenie i obliczyć jego wartość, włącznie z definicjami klas i metod. Jednak odróżnienie składni typowego wyrażenia od typowej instrukcji bywa przydatne. Wyrażenia języka Ruby kontrolujące przepływ sterowania zostały opisane w rozdziale 5. Wyrażenia, które definiują metody i klasy, zostały opisane w rozdziałach 6. i 7.

Niniejszy rozdział opisuje prostszy i bardziej tradycyjny rodzaj wyrażeń. Najprostsze wyrażenia to wartości literalowe, które zostały opisane w rozdziale 3. Ten rozdział opisuje odwołania do zmiennych i stałych, wywołania metod, przypisania oraz wyrażenia złożone z kilku mniejszych wyrażeń połączonych operatorami.

4.1. Literały i literały słów kluczowych

Literał to wartość, taka jak 1.0, 'witajcie' czy [], użyta bezpośrednio w tekście programu. Literały zostały wprowadzone w rozdziale 2., a szczegółowo opisane w rozdziale 3.

Warto wiedzieć, że wiele literalów, jak liczby, są wyrażeniami pierwotnymi — najprostszymi z możliwych, które nie są złożone z żadnych prostszych wyrażeń. Inne literały, jak tablicowe, haszowe czy łańcuchy w podwójnych cudzysłowach używające interpolacji, zawierają podwyrażenia, a więc nie są wyrażeniami pierwotnymi.

Niektóre słowa kluczowe języka Ruby są wyrażeniami pierwotnymi, a więc mogą być uznane za **literały słów kluczowych** lub specjalny rodzaj odwołań do zmiennych:

nil	Daje wartość nil klasy NilClass.
true	Daje wartość będącą singletonowym egzemplarzem klasy TrueClass — obiekt reprezentujący wartość logiczną true.
false	Daje wartość będącą singletonowym egzemplarzem klasy FalseClass — obiekt reprezentujący wartość logiczną false.
self	Daje wartość będącą aktualnym obiektem (więcej informacji na temat tego słowa kluczowego znajduje się w rozdziale 7.).
__FILE__	Daje wartość w postaci łańcucha reprezentującego nazwę wykonywanego przez interpreter pliku. Może być przydatne w komunikatach o błędach.

<u>__LINE__</u>	Daje wartość w postaci liczby całkowitej wyznaczającej numer linii w pliku <u>__FILE__</u> lub aktualnego wiersza kodu.
<u>__ENCODING__</u>	Daje wynik w postaci obiektu klasy <code>Encoding</code> , który określa kodowanie aktualnego pliku (tylko w Ruby 1.9).

4.2. Odwołania do zmiennych

Zmienna to nazwa wartości. Do tworzenia zmiennych i przypisywania im wartości służą wyrażenia przypisania, które zostały opisane dalej. Jeśli nazwa zmiennej pojawia się gdziekolwiek indziej w programie niż po lewej stronie przypisania, jest wyrażeniem odwołania do zmiennej i zostaje zastąpiona swoją wartością:

```
one = 1.0      # Wyrażenie przypisania.
one           # Niniejsze wyrażenie odwołania do zmiennej daje wartość 1.0.
```

Jak pamiętasz z rozdziału 2., w języku Ruby istnieją cztery rodzaje zmiennych; ich nazwy są tworzone według zasad leksykalnych. Zmienne, których nazwy zaczynają się od znaku \$, są zmiennymi globalnymi widocznymi w całym programie. Zmienne obiektowe i klasowe zaczynające się odpowiednio od znaku @ i znaków @@ są używane w programowaniu obiektowym, a ich opis zamieszczony jest w rozdziale 7. W końcu zmienne rozpoczynające się od znaku podkreślenia lub od małej litery są zmiennymi lokalnymi, dostępnymi tylko w obrębie aktualnej metody lub aktualnego bloku (więcej informacji na temat zasięgu zmiennych znajduje się w podrozdziale 5.4.3).

Zmienne zawsze mają proste nazwy bez kwalifikatorów. Jeśli w wyrażeniu znajduje się znak . lub ::, wyrażenie to jest albo odwołaniem do stałej, albo wywołaniem metody. Na przykład wyrażenie `Math::PI` jest odwołaniem do stałej, a wyrażenie `item.price` wywołaniem metody o nazwie `price` na rzecz wartości przechowywanej w zmiennej `item`.

Podczas uruchamiania interpreter Ruby definiuje wstępnie kilka zmiennych globalnych. Ich lista zamieszczona jest w rozdziale 10.

4.2.1. Zmienne niezainicjowane

Z reguły zanim użyje się zmiennej w wyrażeniu, zawsze powinno się najpierw nadać jej wartość, czyli **zainicjować** ją. Są jednak sytuacje, w których Ruby pozwala na używanie niezainicjowanych zmiennych. Zasady dotyczące tej kwestii są inne dla różnych rodzajów zmiennych.

Zmienne klasowe

Zmienne klasowe zawsze muszą mieć przypisaną wartość, aby mogły zostać użyte. Odwołanie do zmiennej klasowej bez przypisanej wartości powoduje zgłoszenie przez Ruby błędu `NameError`.

Zmienne obiektowe

Odwołanie do niezainicjowanej zmiennej obiektowej powoduje zwrot wartości `nil`. Jednak poleganie na tym zachowaniu jest uznawane za zły styl programowania. Ruby zgłasza ostrzeżenie o takiej niezainicjowanej zmiennej, jeśli zostanie uruchomiony z opcją `-w`.

Zmienne globalne

Niezainicjowane zmienne globalne są jak niezainicjowane zmienne obiektowe — dają wartość `nil`, ale powodują zgłoszenie ostrzeżenia, jeśli Ruby zostanie uruchomiony z opcją `-w`.

Zmienne lokalne

Ten przypadek jest bardziej skomplikowany niż poprzednie, ponieważ przed zmiennymi lokalnymi nie ma żadnego prefiku w postaci znaku interpunkcyjnego. Oznacza to, że odwołania do zmiennych lokalnych wyglądają tak samo jak wywołania metod. Jeśli interpreter Ruby spotkał wcześniej przypisanie wartości do zmiennej lokalnej, wie, że nie jest to metoda tylko zmienna, dzięki czemu może zwrócić wartość tej zmiennej. Jeżeli nie było żadnego przypisania, Ruby traktuje takie wyrażenie jako wywołanie metody. W przypadku gdy nie istnieje metoda o takiej nazwie, zgłoszany jest błąd `NameError`.

Ogólnie więc próba użycia zmiennej lokalnej przed jej inicjacją powoduje błąd. Jest jeden wyjątek — zmienna zaczyna istnieć od chwili, w której interpreter napotka wyrażenie przypisania dla tej zmiennej. Ma to miejsce nawet wtedy, gdy operacja przypisania nie zostanie w rzeczywistości wykonana. Istniejąca zmienna nieposiadająca przypisanej żadnej wartości domyślnie dostaje wartość `nil`. Na przykład:

```
a = 0.0 if false      # To przypisanie nie jest nigdy wykonywane.  
print a              # Drukuję nil: zmienna istnieje, ale nie ma przypisanej wartości.  
print b              # NameError: nie ma zmiennej ani metody o nazwie b.
```

4.3. Odwołania do stałych

Stała w języku Ruby jest prawie tym samym co zmienna, z tym wyjątkiem, że jej wartość nie może się zmienić do końca działania programu. Interpreter Ruby nie wymusza, aby stałe nie zmieniały wartości, ale kiedy tak się stanie, zgłasza ostrzeżenie. Z leksykalnego punktu widzenia nazwy stałych wyglądają tak samo jak nazwy zmiennych, ale zaczynają się od wielkich liter. Zgodnie z konwencją nazwy większości stałych są pisane w całości wielkimi literami, a do rozdzielenia poszczególnych słów służą znaki podkreślenia `JAK_TUTAJ`. Nazwy klas i modułów są także stałymi, jednak konwencja nakazuje, aby tylko pierwsza litera każdego wyrazu składowego była wielka `JakTutaj`.

Mimo że stałe wyglądają jak zmienne lokalne z wielkimi literami, mają zasięg zmiennych globalnych — można ich używać w dowolnym miejscu programu. Jednak w przeciwieństwie do zmiennych globalnych stałe mogą być definiowane w klasach i modułach, a więc mogą posiadać nazwy kwalifikowane.

Odwołanie do stałej jest wyrażeniem, którego wartością jest wartość tej stałej. Najprostsze odwołania do stałych są wyrażeniami pierwotnymi — składają się tylko z nazwy stałej:

```
CM_PER_INCH = 2.54    # Definicja stałej.  
CM_PER_INCH          # Odwołanie do stałej. Wartość 2.54.
```

Poza takimi prostymi odwołaniami jak powyżej odwołania do stałych mogą także być złożone. W takim przypadku separatorem nazwy stałej od nazwy klasy lub modułu, do którego on należy, są znaki `::`. Po lewej stronie `::` może znajdować się dowolne wyrażenie, którego wartością jest obiekt klasy lub modułu (jednak zazwyczaj wyrażeniem tym jest zwykłe odwołanie do stałej określające nazwę modułu lub klasy). Po prawej stronie znajduje się nazwa stałej zdefiniowanej w tej klasie lub w tym module. Na przykład:

```
Conversions::CM_PER_INCH # Stała zdefiniowana w module Conversions.  
modules[0]::NAME          # Stała zdefiniowana przez element tablicy.
```

Moduły można zagnieździć, co oznacza, że stałe mogą być definiowane w zagnieżdżonych przestrzeniach nazw jak poniżej:

```
Conversions::Area::HECTARES_PER_ACRE
```

Prawą stronę :: można pominąć. Wtedy stałej poszukuje się w zakresie globalnym:

```
::ARGV      # Stała globalna ARGV.
```

Należy pamiętać, że w rzeczywistości nie ma globalnego zakresu dla stałych. Podobnie jak funkcje globalne, stałe globalne są zdefiniowane (i poszukiwane) w klasie Object. Wyrażenie ::ARGV jest zatem skróconą formą zapisu Object::ARGV.

Kiedy odwołanie do stałej zawiera kwalifikator ::, Ruby wie, gdzie dokładnie jej szukać. Gdy jednak nie ma tego kwalifikatora, interpreter musi znaleźć odpowiednią definicję tej stałej. Przeszukuje zakres leksykalny oraz hierarchię dziedziczenia otaczającej klasy lub modułu. Pełne informacje na ten temat znajdują się w podrozdziale 7.9.

Ruby, zamieniając na wartość wyrażenie odwołania do stałej, zwraca wartość tej stałej lub zgłasza wyjątek NameError, jeśli nie znajdzie żadnej stałej o podanej nazwie. Należy pamiętać, że stała nie istnieje, dopóki nie zostanie jej nadana wartość. Jest to odwrotne zachowanie do zmiennych, które powstają, kiedy interpreter zauważa przypisanie (ale go nie wykona).

Interpreter Ruby definiuje wstępnie kilka stałych podczas uruchamiania. Ich lista znajduje się w rozdziale 10.

4.4. Wywołania metod

Wyrażenie wywołania metody składa się z czterech części:

- Dowolnego wyrażenia, którego wartością jest obiekt, na rzecz którego ma być wywołana metoda. Wyrażenie to jest oddzielone od nazwy metody separatorem . lub ::. Wyrażenie i separator są opcjonalne. Jeśli zostaną pominięte, metoda jest wywoływana na rzecz obiektu wyznaczonego przez słowo kluczowe self.
- Nazwy wywoływanej metody. Jest to jedyna wymagana część wyrażenia wywołania metody.
- Argumentów przekazywanych do metody. Lista argumentów może znajdować się w nawiasach, ale ich stosowanie jest zazwyczaj nieobowiązkowe (opcjonalne i wymagane nawiasy zostały opisane w podrozdziale 6.3). Argumenty rozdzielane są przecinkami. Liczba i typ wymaganych argumentów zależy od definicji metody. Niektóre metody nie przyjmują żadnych argumentów.
- Opcjonalnego bloku kodu otoczonego klamrami lub parą słów do i end. Metoda może wywołać ten kod za pomocą instrukcji yield. Ta możliwość wiązania dowolnego kodu z wywołaniem dowolnej metody jest podstawą niezwykle przydatnych metod iteracyjnych w języku Ruby. Więcej informacji na temat bloków związanych z wywołaniami metod znajduje się w podrozdziałach 5.3 i 5.4.

Nazwa metody jest zazwyczaj oddzielona od obiektu, na rzecz którego została wywołana, znakiem .. Można także używać znaków ::, rzadko jednak korzysta się z tej opcji, ponieważ wtedy wywołania metod wyglądają jak odwołania do stałych.

Kiedy Ruby znajdzie nazwę metody i obiekt, na rzecz którego ta metoda ma zostać wywołana, szuka odpowiedniej definicji tej metody. Proces ten nazywany jest wyszukiwaniem metody lub rozstrzyganiem jej nazwy. Szczegóły tego procesu nie są w tej chwili ważne; można je znaleźć w podrozdziale 7.8.

Wartością wyrażenia wywołania metody jest wartość ostatniego z wyrażeń w ciele tej metody. Więcej informacji na temat definicji, wywoływanego i wartości zwrotnych metod znajduje się w rozdziale 6. Poniżej przedstawionych jest kilka przykładowych wywołań metod:

```
puts "witaj"          # Metoda puts wywołana na rzecz self z jednym argumentem lańcuchowym.  
Math.sqrt(2)          # Metoda sqrt wywołana na rzecz obiektu Math z jednym argumentem.  
message.length        # Metoda length wywołana na rzecz obiektu message; brak argumentów.  
a.each { |x| p x }   # Metoda each wywołana na rzecz obiektu a z dodatkowym blokiem.
```

Wywoływanie funkcji globalnych

Spójrz jeszcze raz na poniższe wywołanie metody:

```
puts "witaj"
```

Jest to wywołanie metody `puts` z modułu `Kernel`. Metody zdefiniowane w tym module są funkcjami globalnymi, podobnie jak wszystkie metody zdefiniowane na najwyższym poziomie poza jakąkolwiek klasą. Funkcje globalne są zdefiniowane jako metody prywatne klasy `Object`. Metody prywatne wprowadzone są w rozdziale 7. Na razie wystarczy wieǳia, że metody prywatne nie mogą być jawnie wywoływane na rzecz obiektu — są zawsze wywoływane niejawnie na rzecz `self`. Obiekt `self` jest zawsze zdefiniowany i bez względu na jego wartość jest on zawsze klasy `Object`. Ponieważ funkcje globalne są metodami klasy `Object`, mogą one być wywoływane zawsze (niejawnie) w każdym miejscu bez względu na wartość `self`.

Jednym z prezentowanych wcześniej wywołań metod było `message.length`. Można poczuć pokusę, aby traktować je jako odwołanie do zmiennej, którego wartością jest wartość zmiennej `length` w obiekcie `message`. Nie jest to jednak prawda. Ruby posiada bardzo czysty model obiektowy — obiekty w tym języku mogą zawierać dowolną liczbę wewnętrznych zmiennych obiektowych, ale do użytku na zewnątrz mogą udostępniać tylko metody. Ponieważ metoda `length` nie pobiera żadnych argumentów i jest wywoływana bez opcjonalnych nawiasów, wygląda jak odwołanie do zmiennej. W rzeczywistości takie było zamierzenie projektanta. Metody tego typu nazywają się metodami dostępu, a `length` to atrybut obiektu `message`¹. Obiekt `message` może definiować metodę o nazwie `length=`. Jeśli pobiera ona jeden argument, jest metodą ustawiającą atrybut i Ruby będzie wywoływał ją w odpowiedzi na przypisania. Gdy taka metoda jest zdefiniowana, poniższe dwa wiersze kodu wywołają tę samą metodę:

```
message.length=(3)      # Zwykłe wywołanie metody.  
message.length = 3       # Wywołanie metody wyglądające jak przypisanie.
```

Spójrz teraz na poniższy wiersz kodu przy założeniu, że zmienna `a` przechowuje tablicę:

```
a[0]
```

¹ Nie oznacza to, że każda metoda niepobierająca argumentów daje dostęp do atrybutów. Na przykład metoda tablicowa `sort` nie pobiera argumentów, ale nie można powiedzieć, że zwraca wartość jakiegoś atrybutu.

Ponownie można mylnie stwierdzić, że jest to jakiś specjalny rodzaj odwołania do zmiennej, gdzie zmienna jest w rzeczywistości elementem tablicy. Jest to jednak wywołanie metody. Interpreter Ruby konwertuje dostęp do tablicy następująco:

```
a.[](0)
```

Wyrażenie dostępu do tablicy zamienia się w wywołanie metody o nazwie `[]` na rzecz tej tablicy, a jej argumentem jest indeks tablicy. Zastosowanie tej składni dostępu nie ogranicza się tylko do tablic. Każdy obiekt może definiować metodę o nazwie `[]`. Kiedy obiekt jest „indeksowany” za pomocą nawiasów kwadratowych, wszystkie wartości w nich się znajdujące zostaną przekazane do tej metody. Jeśli metoda `[]` pobiera trzy argumenty, w nawiasach kwadratowych powinny znajdować się trzy wyrażenia oddzielone przecinkami.

Przypisanie do tablicy jest także wykonywane przez wywołanie metody. Jeżeli obiekt o definiuje metodę o nazwie `[]=`, wyrażenie `o[x]=y` jest zamieniane na `o.[]=(<x>, <y>)`, a wyrażenie `o[x,y]=z` na `o.[]=(<x>, <y>, <z>)`.

Dalsza część tego rozdziału zawiera informacje, że wiele operatorów w języku Ruby jest zdefiniowanych jako metody, dzięki czemu wyrażenia typu `x+y` są obliczane jako `x.+(<y>)`, gdzie nazwa metody to `+`. Dzięki temu że wiele operatorów Ruby jest zdefiniowanych jako metody, operatory te można przeddefiniowywać we własnych klasach.

Teraz przyjrzyj się następującemu prostemu wyrażeniu:

```
x
```

Jeżeli istnieje zmienna o nazwie `x` (to znaczy jeśli interpreter Ruby znalazł przypisanie wartości do `x`), jest to wyrażenie odwołania do zmiennej. W przypadku gdy zmienna taka nie istnieje, jest to wywołanie metody `x` bez żadnych argumentów na rzecz `self`.

Specjalnym wyrażeniem wywołania metody jest zarezerwowane słowo Ruby `super`. Jest ono używane przy tworzeniu podklasy innej klasy. Słowo kluczowe `super` przekazuje argumenty bieżącej metody do metody o tej samej nazwie w nadklasie. Można go również używać tak, jakby było metodą; przyjmuje ono nawet listę argumentów. Słowo `super` zostało szczegółowo opisane w podrozdziale 7.3.3.

4.5. Przypisywanie

Wyrażenie przypisania określa przynajmniej jedną wartość dla przynajmniej jednej l-wartości. **L-wartość** (ang. *lvalue*) to termin opisujący wszystko, co może występować po lewej stronie operatora przypisania (wartości po jego prawej stronie są czasami dla odróżnienia nazywane **r-wartościami** od ang. *rvalue*). W języku Ruby l-wartościami są zmienne, stałe, atrybuty i elementy tablic. Zasady i znaczenie wyrażeń przypisania dla różnych rodzajów l-wartości są nieco inne. Każdy z tych rodzajów został opisany w niniejszym podrozdziale.

W Ruby istnieją trzy rodzaje wyrażeń przypisania. Proste przypisanie zawiera jedną l-wartość, operator `=` i r-wartość. Na przykład:

```
x = 1      # Ustawienie wartości x na 1.
```

Przypisanie skrócone jest skróconym wyrażeniem aktualizującym wartość zmiennej za pomocą jakiejś innej operacji (jak dodawanie) wykonywanej na rzecz bieżącej wartości zmiennej. Przypisanie skrócone wykorzystuje operatory przypisania typu `+=` i `*=`, które powstały z połączenia operatorów binarnych ze znakiem równości:

```
x += 1      # Ustawienie l-wartości x na wartość x + 1.
```

W końcu przypisanie równolegle to takie wyrażenie przypisania, w którym występuje więcej niż jedna l-wartość lub więcej niż jedna r-wartość. Oto prosty przykład:

```
x, y, z = 1, 2, 3 # Ustawienie x na 1, y na 2 i z na 3.
```

Przypisanie równolegle komplikuje się, gdy liczba l-wartości nie odpowiada liczbie r-wartości lub gdy po prawej stronie znajduje się tablica. Szczegóły na ten temat znajdują się poniżej.

Wartością wyrażenia przypisania jest przypisana wartość (lub tablica wartości). Dodatkowo operator przypisania jest operatorem o łączności prawostronnej — jeśli wyrażenie zawiera kilka przypisań, są one wykonywane od prawej do lewej. Oznacza to, że przypisania można łączyć w łańcuchy mające na celu przypisanie tej samej wartości do kilku zmiennych:

```
x = y = 0 # Ustawienie x i y na 0.
```

Należy zauważyć, że nie jest to przypisanie równolegle — są to dwa proste przypisania połączone w łańcuch. Zmiennej *y* jest przypisywana wartość 0, a zmiennej *x* — wartość (również 0) tego pierwszego przypisania.

Efekty uboczne przypisania

Ważniejszy od wartości wyrażenia przypisania jest fakt, że przypisania ustawiają wartość zmiennej (lub innej l-wartości), a więc mają wpływ na stan programu. Ten wpływ jest nazywany **efektem ubocznym** (ang. *side effect*) przypisania.

Wiele wyrażeń nie ma efektu ubocznego, a więc nie wpływa na stan programu. Wyrażenia te są **idempotentne** (niepowtarzalne). Oznacza to, że wyrażenie takie może być obliczane wiele razy i za każdym razem zwróci tę samą wartość. Znaczy to również, że obliczenie wartości takiego wyrażenia nie ma wpływu na wartości innych wyrażeń. Poniżej znajdują się przykładowe wyrażenia niemające efektu ubocznego:

```
x + y  
Math.sqrt(2)
```

Ważne jest, aby zrozumieć, że przypisania nie są idempotentne:

```
x = 1      # Wpływ na wartości innych wyrażeń, które używają zmiennej x.  
x += 1    # Zwraca za każdym razem inną wartość.
```

Niektóre metody, na przykład `Math.sqrt`, są idempotentne — mogą być wywoływane bez żadnych efektów ubocznych. Inne metody nie są. W dużym stopniu zależy to od tego, czy metody te dokonują przypisań do zmiennych nielokalnych.

4.5.1. Przypisania do zmiennych

Słowo „przypisanie” przywołuje na myśl zmienne, ponieważ są one najczęściej spotykana l-wartością w wyrażeniach przypisania. Przypomnijmy, że język Ruby posiada cztery rodzaje zmiennych: lokalne, globalne, obiektowe i klasowe. Różnią się one pierwszym znakiem w nazwie. Przypisanie działa tak samo w przypadku wszystkich czterech rodzajów zmiennych, a więc w tych operacjach ich rozróżnianie jest niepotrzebne.

Należy pamiętać, że zmienne obiektowe w Ruby nigdy nie są widoczne poza swoim obiektem, a więc nazwy zmiennych nigdy nie posiadają kwalifikatora w postaci nazwy obiektu. Przeanalizuj poniższe przypisanie:

```
point.x, point.y = 1, 2
```

L-wartości w tym przypisaniu nie są zmiennymi. Są to atrybuty, których opis znajduje się nieco dalej.

Przypisanie do zmiennej odbywa się zgodnie z oczekiwaniami — zmienna jest ustawiana na wyznaczoną wartość. Jedyny problem dotyczy deklaracji zmiennych i niejednoznaczności między nazwami zmiennych lokalnych a nazwami metod. Język Ruby nie posiada żadnej składni pozwalającej jawnie zadeklarować zmienną — zmienne zaczynają istnieć w chwili przypisania im wartości. Dodatkowo nazwy zmiennych lokalnych i metod wyglądają tak samo — nie ma żadnego przedrostka typu \$ do ich odróżnienia. Dlatego proste wyrażenie typu x może być odwołaniem do zmiennej lokalnej o nazwie x lub metodą self o tej samej nazwie. Aby rozwiązać ten problem dwuznaczności, Ruby traktuje identyfikator jako zmienią lokalną, jeśli wcześniej spotkał przypisanie do tej zmiennej. Robi to nawet wówczas, gdy przypisanie to nigdy nie zostało wykonane. Demonstruje to poniższy fragment kodu:

```
class Ambiguous
  def x; 1; end # Metoda o nazwie x. Zawsze zwraca wartość 1.
  def test
    puts x      # Nie znaleziono żadnej zmiennej; odwołuje się do powyższej metody i drukuje 1.
    # Wartość poniższego wiersza kodu nie jest nigdy obliczana ze względu na klawisz if false.
    # Jednak interpreter widzi go i traktuje x jako zmienną w pozostałej części tej metody.
    x = 0 if false
    puts x      # x jest zmienną, ale nigdy nie przypisano mu wartości — drukuje nil.
    x = 2      # Niniejsze przypisanie zostanie obliczone.
    puts x      # Dlatego niniejsza instrukcja wydrukuję 2.
  end
end
```

4.5.2. Przypisania do stałych

Różnica pomiędzy stałymi a zmiennymi jest oczywista — wartość stałej powinna z założenia pozostać niezmieniona przez cały czas działania programu. Dlatego przypisania do stałych podlegają kilku specjalnym zasadom:

- Przypisanie wartości stałej, która już istnieje, powoduje zgłoszenie przez Ruby ostrzeżenia. Jednak samo takie przypisanie zostaje wykonane, co oznacza, że stałe nie są w rzeczywistości stałe.
- Nie można przypisywać wartości stałym w ciele metod. Ruby zakłada, że metody są przeznaczone do wielokrotnego użytku. Gdyby w ciele metody można było wstawać przypisania do stałych, każde wywołanie takiej metody, oprócz pierwszego, powodowałoby zgłoszenie ostrzeżenia. Dlatego działanie takie jest zabronione.

W przeciwnieństwie do zmiennych stałe nie powstają, dopóki interpreter nie wykona wyrażenia przypisania. Nieobliczane wyrażenie, które znajduje się poniżej, **nie** powoduje powstania stałej:

```
N = 100 if false
```

Wynika z tego, że stała nie może być niezainicjowana. Jeśli stała istnieje, to musi mieć wartość. Stała może mieć wartość nil tylko wówczas, gdy zostanie jej ona celowo nadana.

4.5.3. Przypisania do atrybutów i elementów tablicy

Przypisanie do atrybutu i elementu tablicy jest w języku Ruby w rzeczywistości skróconą formą wywołania metody. Założmy, że obiekt o udostępnia metodę o nazwie `m=` — na końcu nazwy tej metody jest znak równości. W takiej sytuacji po lewej stronie wyrażenia przypisania można wpisać `o.m`. Dodatkowo założmy, że przypisujesz wartość v:

```
o.m = v
```

Interpreter Ruby konwertuje to przypisanie na poniższe wywołanie metody:

```
o.m=(v) #Jeśli zostaną opuszczone nawiasy i dodana spacja, wyrażenie to będzie wyglądać jak przypisanie!
```

To znaczy że wartość `v` zostaje przekazana do metody `m=`, która może wykonać dowolne działanie na tej wartości. Z reguły jest to sprawdzenie, czy wartość jest odpowiedniego typu i mieści się w odpowiednim zakresie, a następnie zapisanie w zmiennej obiektowej obiektu. Metodom typu `m=` zazwyczaj towarzyszy metoda `m`, która zwraca ostatnią wartość przekazaną do `m=`. Metody typu `m=` nazywane są metodami **ustawiającymi** (setters), a metody typu `m` metodami **dostępowymi** (getters). Kiedy obiekt udostępnia obie te metody, mówi się, że ma atrybut `m`. W innych językach atrybuty bywają nazywane własnościami lub właściwościami. Więcej informacji na temat atrybutów znajduje się w podrozdziale 7.1.5.

Przypisywanie wartości do elementów tablic także odbywa się za pomocą wywołań metod. Jeśli obiekt `o` definiuje metodę `o` nazwie `[]=` (nazwa metody składa się tylko z tych trzech znaków), która pobiera dwa argumenty, wyrażenie `o[x]=y` jest w rzeczywistości wykonywane jako:

```
o.[]= (x, y)
```

Gdy obiekt udostępnia metodę `[]=` pobierającą trzy argumenty, może być indeksowany przy użyciu dwóch wartości umieszczonych w nawiasach kwadratowych. Poniższe dwa wyrażenia są w tym przypadku równoważne:

```
o[x, y] = z  
o.[]= (x, y, z)
```

4.5.4. Skrócone przypisania

Skrócone przypisanie to takie, które jest połączone z jakąś inną operacją. Najczęściej jest używane do zwiększania wartości zmiennych:

```
x += 1
```

Znaki `+=` nie są prawdziwym operatorem w języku Ruby. Powyższe wyrażenie jest skróconą formą zapisu następującego wyrażenia:

```
x = x + 1
```

Skróconego przypisania nie można połączyć z przypisaniem równoległy — działa tylko wówczas, gdy po lewej i prawej stronie znajduje się tylko po jednej wartości. Nie należy go używać, kiedy po lewej stronie znajduje się stała, ponieważ powoduje to ponowne przypisanie wartości, co z kolei prowadzi do ostrzeżenia. Przypisania skróconego można natomiast używać, kiedy wartość po lewej stronie jest atrybutem. Poniższe dwa wyrażenia są równoważne:

```
o.m += 1  
o.m=(o.m()+1)
```

Skrócone przypisanie działa nawet wówczas, gdy wartość po lewej stronie jest elementem tablicy. Poniższe dwa wyrażenia są równoważne:

```
o[x] -= 2  
o.[]= (x, o.[](x) - 2)
```

W powyższym kodzie użyto przypisania `-=` zamiast `+=`. Jak nie trudno się domyślić, pseudoperator `-` odejmuje wartość znajdująca się po prawej stronie od wartości z lewej strony.

Poza `+=` i `-=` istnieje jeszcze jedenaście innych pseudooperatorów, które mogą być używane jako skrócone przypisania. Listę tych operatorów przedstawia tabela 4.1. Należy pamiętać, że nie są to operatory same w sobie, tylko skrócone formy wyrażeń wykorzystujących operatory. Działanie tych pozostałych operatorów zostało opisane dalej w tym rozdziale. Ponadto w dalszej części znajdują się informacje o tym, że wiele z tych operatorów jest zdefiniowanych jako metody. Jeśli na przykład klasa definiuje metodę o nazwie `+`, to zmienia się działanie skróco-nych przypisań z użyciem operatora `+=` we wszystkich egzemplarzach tej klasy.

Tabela 4.1. Skrócone pseudooperatory przypisania

Przypisanie	Rozwinięcie
<code>x += y</code>	<code>x = x + y</code>
<code>x -= y</code>	<code>x = x - y</code>
<code>x *= y</code>	<code>x = x * y</code>
<code>x /= y</code>	<code>x = x / y</code>
<code>x %= y</code>	<code>x = x % y</code>
<code>x **= y</code>	<code>x = x ** y</code>
<code>x &&= y</code>	<code>x = x && y</code>
<code>x = y</code>	<code>x = x y</code>
<code>x &= y</code>	<code>x = x & y</code>
<code>x = y</code>	<code>x = x y</code>
<code>x ^= y</code>	<code>x = x ^ y</code>
<code>x <<= y</code>	<code>x = x << y</code>
<code>x >>= y</code>	<code>x = x >> y</code>

Idiom `||=`

Jak napisano na początku niniejszego rozdziału, skrócone przypisania najczęściej znajdują zastosowanie w zwiększeniu wartości zmiennych za pomocą pseudooperatora `+=`. Często też zmniejsza się wartość zmiennych przy użyciu operatora `-=`. Pozostałe pseudooperatory są używane znacznie rzadziej. Warto jednak wiedzieć o jednym idiomie. Założymy, że piszesz metodę obliczającą jakieś wartości, wstawiającą je do tablicy i zwracającą tę tablicę. Chcesz umożliwić użytkownikowi wybranie, do której tablicy wartości te mają być wstawiane. Jeśli użytkownik nie określi tablicy, nie tworzysz nowej. Do tego celu może być przydatny poniższy wiersz kodu:

```
results ||= []
```

Jego rozwinięcie jest następujące:

```
results = results || []
```

Osoby, które znają operator `||` z innych języków lub przeczytały o nim dalej w tej książce, wiedzą, że wartością prawej strony tego przypisania jest wartość zmiennej `results`, chyba że jest to `nil` albo `false`. W takim przypadku wartością jest nowa pusta tablica. Oznacza to, że prezentowane tutaj skrócone przypisanie pozostawia zmienną `results` bez zmian, pod warunkiem że nie ma ona wartości `nil` ani `false` — wtedy zostałaby przypisana nowa tablica.

Skrócony operator przypisania `||=` działa w rzeczywistości nieco inaczej niż pokazane rozwinięcie. Jeśli po jego lewej stronie nie ma wartości `nil` ani `false`, nic nie jest przypisywane. Jeżeli wartością po lewej stronie jest atrybut lub element tablicy, metoda ustawiająca wykonująca przypisanie nie zostaje wywołana.

4.5.5. Przypisanie równoległe

Przypisaniem równoległy jest każde wyrażenie przypisania, po którego lewej lub prawej stronie albo po obu stronach znajduje się więcej niż jedna wartość; są one oddzielane przecinkami. Wartości z obu stron mogą mieć przedrostek * — jest to tzw. operator **splat**, aczkolwiek nie jest on prawdziwym operatorem. Działanie * jest opisane dalej.

Większość wyrażeń przypisania równoległego jest prosta, a ich znaczenie jest oczywiste. Są jednak także skomplikowane przypadki, które zostały opisane w kolejnych podrozdziałach.

4.5.5.1. Taka sama liczba wartości po lewej i prawej stronie

Przypisanie równoległe w najprostszej formie ma miejsce wówczas, gdy po prawej i lewej stronie znajduje się taka sama liczba wartości:

```
x, y, z = 1, 2, 3    # x=1; y=2; z=3
```

W tym przypadku pierwsza wartość z prawej strony jest przypisywana do pierwszej wartości z lewej strony, druga do drugiej, a trzecia do trzeciej.

Przypisania te są wykonywane równolegle, a nie sekwencyjnie. Na przykład poniższe dwa wiersze kodu nie są równoważne:

```
x, y = y, x      # Równolegle — zamiana wartości dwóch zmiennych.  
x = y; y = x    # Sekwencyjne — obie zmienne mają tę samą wartość.
```

4.5.5.2. Jedna wartość po lewej i kilka wartości po prawej stronie

Kiedy po lewej stronie jest jedna wartość, a po prawej kilka, tworzona jest tablica wartości z prawej strony, która zostaje przypisana do wartości z lewej strony:

```
x = 1, 2, 3      # x = [1,2,3].
```

Przed wartością z lewej strony można postawić znak *. Nie spowoduje to zmiany znaczenia ani wartości zwrotnej tego przypisania.

Aby zapobiec tworzeniu tablicy z wartości po prawej stronie, po wartości z lewej strony należy wstawić przecinek. Nawet jeśli po przecinku tym nie będzie żadnej wartości, Ruby postrahuje to tak, jakby po lewej stronie było ich kilka:

```
x, = 1, 2, 3      # x = 1; pozostałe wartości są odrzucone.
```

4.5.5.3. Kilka wartości po lewej, jedna tablica po prawej stronie

Kiedy jest kilka wartości po lewej stronie, a tylko jedna wartość po prawej, Ruby próbuje rozłożyć wartość z prawej strony na listę wartości do przypisania. Jeśli po prawej stronie jest tablica, zostaje ona rozłożona, tak że każdy element staje się osobną wartością. W sytuacji gdy wartość z prawej strony nie jest tablicą, ale implementuje metodę `to_ary`, Ruby wywołuje tę metodę i rozkłada zwróconą przez nią tablicę:

```
x, y, z = [1, 2, 3]  # To samo co x,y,z = 1,2,3.
```

Przypisanie równoległe zostało tak przekonwertowane, że po lewej stronie znajduje się kilka wartości, a po prawej zero (jeśli rozbita tablica była pusta) lub więcej wartości. Gdy liczba wartości po lewej stronie odpowiada liczbie wartości po prawej, następuje przypisanie zgodne z zasadami opisanymi w podrozdziale 4.5.5.1. Jeżeli liczby te nie zgadzają się, przypisanie jest zgodne z zasadami podanymi w podrozdziale 4.5.5.4.

Aby przetransformować zwykłe nierównoległe przypisanie na równoległe, można zastosować opisaną wyżej sztuczkę z przecinkiem, która powoduje automatyczne rozpakowanie tablicy po prawej stronie:

```
x = [1,2]      # x ma wartość [1,2]: to nie jest przypisanie równolegle.  
x, = [1,2]    # x ma wartość 1: przecinek na końcu sprawia, że jest to przypisanie równolegle.
```

4.5.5.4. Różne liczby wartości po obu stronach

Jeśli wartości po lewej stronie jest więcej niż po prawej i nie ma żadnego operatora splat, pierwsza wartość z prawej strony jest przypisywana do pierwszej z lewej, druga z prawej do drugiej z lewej itd. aż do wyczerpania wartości z prawej strony. Pozostałym wartościom z lewej strony jest przypisywana wartość `nil`, która nadpisuje istniejącą wartość:

```
x, y, z = 1, 2 #x=1; y=2; z=nil.
```

W przypadku gdy więcej jest wartości z prawej niż z lewej strony i nie ma żadnego operatora splat, wartości z prawej są przypisywane w takiej kolejności, w jakiej są wpisane, do wartości z lewej, a pozostałe wartości z prawej strony są odrzucane:

```
x, y = 1, 2, 3 #x=1; y=2; 3 nie zostaje przypisana nigdzie.
```

4.5.5.5. Operator splat

Jeżeli przed wartością z prawej strony znajduje się gwiazdka, znaczy to, że wartość ta jest tablicą (lub obiektem podobnym do tablicy) i że każdy jej element powinien być osobną wartością. Elementy tablicy zastępują pierwotną tablicę; następuje przypisanie zgodne z opisanymi wyżej zasadami:

```
x, y, z = 1, *[2,3] # To samo co x,y,z = 1,2,3.
```

W Ruby 1.8 operator splat może znajdować się wyłącznie przed ostatnią wartością z prawej strony przypisania. W Ruby 1.9 po prawej stronie przypisania równoległego może znajdować się dowolna liczba operatorów splat i mogą one występować w dowolnym miejscu na liście. W obu wersjach języka zabronione jest jednak stawianie dwóch gwiazdek przed zagnieżdzoną tablicą:

```
x,y = **[[1,2]] # SyntaxError!
```

W Ruby gwiazdkę można stawiać przed tablicami, zakresami i tablicami asocacyjnymi występującymi po prawej stronie przypisania. W ogóle każda r-wartość definiująca metodę `to_a` może być poprzedzona operatorem splat. Każdy obiekt `Enumerable`, w tym wyliczenia (zobacz podrozdział 5.3.4), może być poprzedzony gwiazdką. Jeśli gwiazdka zostanie postawiona przed wartością, która nie definiuje metody `to_a`, wartość ta zostanie zastąpiona przez siebie samą.

Jeśli gwiazdka znajduje się przed wartością z lewej strony, wszystkie niepotrzebne wartości z prawej strony są wstawiane do tablicy i przypisywane do tej l-wartości — są one zawsze tablicami zawierającymi zero, jeden lub więcej elementów:

```
x,*y = 1, 2, 3 #x=1; y=[2,3]  
x,*y = 1, 2      #x=1; y=[2]  
x,*y = 1        #x=1; y=[]
```

W Ruby 1.8 gwiazdka może znajdować się tylko przed ostatnią wartością z lewej strony. W Ruby 1.9 również po lewej stronie może znajdować się tylko jedna gwiazdka, ale w dowolnym miejscu:

```
# Tylko Ruby 1.9.  
*x, y = 1, 2, 3  #x=[1,2]; y=3.  
*x, y = 1, 2    #x=[1]; y=2.  
*x, y = 1       #x=[]; y=1.
```

Należy zauważać, że gwiazdki mogą pojawiać się po obu stronach wyrażenia przypisania:

```
x, y, *z = 1, *[2, 3, 4] #x=1; y=2; z=[3,4].
```

Na końcu przypomnijmy, że wcześniej były opisywane dwa proste przypadki wyrażeń przypisania, w których brała udział tylko jedna wartość z lewej strony lub jedna wartość z prawej strony. Warto zauważać, że obie te sytuacje są takie same, jak gdyby przed tymi wartościami znajdowała się gwiazdka. Jawne dodanie tego znaku w obu wymienionych przypadkach nie da żadnego dodatkowego rezultatu.

4.5.5.6. Nawiasy w przypisaniu równoległy

Jedną z naj słabiej rozumianych własności przypisania równoległego jest możliwość używania po lewej stronie nawiasów służących do tworzenia podprzypisań. Jeśli grupa dwóch lub więcej l-wartości znajduje się w nawiasach, jest początkowo traktowana jako pojedyncza wartość. Po znalezieniu odpowiadającej jej r-wartości stosowane są rekursywne reguły przypisania równoległego — r-wartość zostaje przypisana do grupy l-wartości, która znajdowała się w nawiasach. Spójrz na poniższy przykład:

```
x, (y, z) = a, b
```

W rzeczywistości wykonywane są dwie operacje przypisania jednocześnie:

```
x = a  
y, z = b
```

Należy jednak zauważać, że drugie przypisanie samo jest przypisaniem równoległym. Dzięki użyciu po lewej stronie nawiasów zostało wykonane rekursywne przypisanie równolegle. Aby to działało, b musi być obiektem akceptującym gwiazdkę, jak tablica lub wyliczenie.

Oto kilka konkretnych przykładów, które powinny wszystko wyjaśnić. Należy zauważać, że nawiasy po lewej stronie odpakowują jeden poziom zagnieżdżonej tablicy po prawej:

```
x, y, z = 1, [2, 3]          # Brak nawiasów: x=1; y=[2,3]; z=nil.  
x, (y, z) = 1, [2, 3]        # Nawiasy: x=1; y=2; z=3.  
a, b, c, d = [1, [2, [3, 4]]] # Brak nawiasów: a=1; b=2, [3,4]]; c=d=nil.  
a, (b, (c, d)) = [1, [2, [3, 4]]] # Nawiasy: a=1; b=2; c=3; d=4.
```

Przypisanie równolegle i wywoływanie metod

Na marginesie warto zauważać, że jeśli przed przypisaniem równoległym znajduje się nazwa metody, interpreter potraktuje przecinki jako separatory argumentów metody, a nie separatory l-wartości lub r-wartości. Aby sprawdzić wartość zwrotną przypisania równoległego, można napisać następującą procedurę drukującą:

```
puts x, y=1, 2
```

Kod ten nie działa jednak zgodnie z oczekiwaniemi. Ruby wnioskuje, że metoda puts jest wywoływana z trzema argumentami — x, y=1 i 2. Następnie można spróbować wstawić przypisanie równolegle do nawiasów w celu utworzenia grupy:

```
puts (x, y=1, 2)
```

4.5.5.7. Wartość przypisania równoległego

Wartość zwrotna wyrażenia przypisania równoległego jest tablicą r-wartości (po wzbogaceniu operatorami splat).

4.6. Operatory

Operator to token reprezentujący jakąś operację (na przykład dodawanie czy porównywanie), która jest wykonywana na rzecz jednego lub większej liczby operandów. Operandy są wyrażeniami, a operatory pozwalają na łączenie ich w jeszcze większe wyrażenia. Na przykład literal liczbowy 2 i operator + można połączyć w wyrażenie 2 + 2. Poniższe wyrażenie składa się z literalu liczbowego, wyrażenia wywołującego metodę, odwołania do zmiennej oraz operatorów mnożenia i mniejszości:

```
2 * Math.sqrt(2) < limit
```

Tabela 4.2 zawiera zestawienie wszystkich operatorów języka Ruby. Każdy z nich został szczegółowo opisany w kolejnych podrozdziałach. Aby jednak w pełni zrozumieć działanie operatorów, należy zapoznać się z takimi zagadnieniami, jak krotkość (ang. *arity*), priorytetowość (ang. *precedence*) oraz łączność (ang. *associativity*).

Krotkość operatora to liczba operandów, na których może on operować. Operatory jednoargumentowe pobierają tylko jeden operand. Operatory binarne (dwuargumentowe) przyjmują dwa operandy, a trójargumentowe (jest tylko jeden taki) trzy. Krotkość operatora w tabeli 4.2 jest zaznaczona w kolumnie *N*. Należy zauważyć, że operatory + i - mają zarówno formę jedno-, jak i dwuargumentową.

Priorytetowość operatora określa, jak mocno jest on przywiązyany do swoich operandów, i ma wpływ na kolejność wykonywania działań w wyrażeniu. Weźmy na przykład poniższe wyrażenie:

```
1 + 2 * 3      #=> 7.
```

Operator mnożenia ma wyższy priorytet od operatora dodawania, dlatego mnożenie jest wykonywane jako pierwsze, a zwrócona wartość to 7. Operatory w tabeli 4.2 są ustawione w kolejności od priorytetu najwyższego do najniższego. Warto zauważyć, że operatory operacji logicznych AND, OR i NOT występują zarówno w wersji o wysokim, jak i niskim priorytecie.

Priorytetowość operatorów określa tylko domyślną kolejność wykonywania działań w wyrażeniu. Kolejność wykonywania poszczególnych części każdego wyrażenia można zmienić za pomocą nawiasów. Na przykład:

```
(1 + 2) * 3      #=> 9.
```

Łączność operatora określa kolejność wykonywania działań w sytuacji, kiedy ten sam operator (lub operatory o takich samych priorytetach) występuje więcej niż jeden raz po kolei. Łączność operatorów w tabeli 4.2 jest oznaczona w kolumnie *L*. Wartość *P* — działania wykonywane są od prawej do lewej, wartość *N* — operator nie jest łączny, a więc nie może występować kilka razy w wyrażeniu bez nawiasów, które określają kolejność wykonywania działań.

Tabela 4.2. Operatory języka Ruby ułożone według priorytetów z oznaczoną krotnością (K), łącznością (L) i możliwością przeddefiniowania (M)

Operator(y)	K	L	M	Działanie
! ~ +	1	P	T	Logiczna operacja NOT, dopełnienie bitowe, jednoargumentowy plus ²
**	2	P	T	Potęgowanie
-	1	P	T	Jednoargumentowy minus (należy definiować ze znakiem @)
* / %	2	L	T	Mnożenie, dzielenie, modulo (reszta z dzielenia)
+ -	2	L	T	Dodawanie (lub konkatenacja), odejmowanie
<< >>	2	L	T	Przesunięcie bitowe w lewo (lub dołączenie), przesunięcie bitowe w prawo
&	2	L	T	Bitowa operacja AND
^	2	L	T	Bitowa operacja OR, bitowa operacja XOR
< <= >= >	2	L	T	Porządkowanie
== === != =~ !~ <=>	2	N	T	Równość, dopasowywanie wzorców, porównywanie ³
&&	2	L	N	Operacja logiczna AND
	2	L	N	Operacja logiczna OR
... ...	2	N	N	Tworzenie zakresów i przerzutniki logiczne
?:	3	P	N	Operacja warunkowa
rescue	2	L	N	Modyfikator obsługi wyjątków
= **= *= /= %= += -= <<=	2	P	N	Przypisanie
>>= &&= \$= = = ^=				
defined?	1	N	N	Sprawdzanie definicji i typu zmiennej
not	1	P	N	Logiczna operacja NOT (niski priorytet)
and or	2	L	N	Logiczna operacja AND, logiczna operacja OR (niski priorytet)
if unless while until	2	N	N	Modyfikatory warunkowe i pętle

Większość operatorów arytmetycznych jest łącznych lewostronnie, co oznacza, że wyrażenie 10-5-2 jest równoważne z (10-5)-2, a nie 10-(5-2). Natomiast potęgowanie jest łączne prawostronne. Zatem $2^{**}3^{**}4$ jest równoważne z $2^{**}(3^{**}4)$. Innym przykładem operatora z łącznością prawostronną jest operator przypisania. W wyrażeniu $a=b=0$ wartość 0 jest najpierw przypisywana zmiennej b. Następnie wartość tego wyrażenia (także 0) jest przypisywana do zmiennej a.

Niektóre operatory zostały zaimplementowane jako metody, dzięki czemu można je przeddefiniować w wybranych klasach. Informacje dotyczące tego, czy operator jest metodą, znajdują się w kolumnie M w tabeli 4.2. Wartość T oznacza, że operator jest metodą i można go przeddefiniować, a wartość N — że tak nie jest. W zasadzie klasy mogą definiować własną arytmetykę, kolejność i równość operatorów, ale nie mogą przeddefiniować różnych operatorów logicznych. Prezentowane operatory podzielone zostały według przydatności w standardowych klasach Ruby. W innych klasach operatory te mogą działać inaczej. Na przykład operator + dodaje wartości liczbowe i jest sklasyfikowany jako operator arytmetyczny. Jest

² Operator ! może być przeddefiniowywany od wersji Ruby 1.9. Jednoargumentowy plus należy definiować ze znakiem @.

³ Operatory != i =~ można przeddefiniowywać od wersji Ruby 1.9.

jednak także używany do konkatenacji łańcuchów i tabel. Operator będący metodą jest wywoływany na rzecz swojego lewego argumentu (lub jedynego w przypadku operatorów jednoargumentowych). Operand znajdujący się po prawej stronie jest przekazywany do tej metody jako argument. Definicję operatora zaimplementowanego jako metoda można sprawdzić tak samo jak definicję każdej innej metody klasowej. Na przykład poniżej sprawdzona zostanie definicja operatora * w klasie String za pomocą narzędzia ri:

```
ri 'String.*'
```

Definiując jednoargumentowe operatory + i -, należy zastosować nazwy metod +@ i -@, co pozwoli uniknąć pomylenia ich z operatorami dwuargumentowymi, które są reprezentowane przez te same symbole. Operatory != i !~ są zdefiniowane jako negacja operatorów == i =~. W Ruby 1.9 istnieje możliwość przedefiniowania operatorów != i !~. We wcześniejszych wersjach tego języka nie było to możliwe. Ruby 1.9 pozwala także na przedefiniowanie jednoargumentowego operatora !.

4.6.1. Jednoargumentowe operatory + i -

Jednoargumentowy operator - zmienia znak swojego argumentu liczbowego. Stosowanie jednoargumentowego operatora + jest także dozwolone, ale nie ma on żadnego wpływu na operandy liczbowe — zwraca wartość swojego operandu. Jest udostępniany w celu zachowania symetrii z jednoargumentowym minusem i można go oczywiście przedefiniowywać. Warto zauważyć, że jednoargumentowy minus ma nieco niższy priorytet od jednoargumentowego plusa. Informacje na ten temat znajdują się w kolejnym podrozdziale na temat operatora **.

Nazwy tych operatorów jako metod to -@ i +@. Należy ich używać przy przedefiniowywaniu tych operatorów jako metod, wywoływanie ich jako metod lub szukaniu ich dokumentacji. Te specjalne nazwy są niezbędne do odróżnienia jednoargumentowych wersji tych operatorów od wersji dwuargumentowych.

4.6.2. Potęgowanie: **

Operator ** wykonuje potęgowanie, czyli podnosi swój pierwszy argument do potęgi określonej drugim argumentem. Wstawiając w miejsce drugiego operandu ułamek, można obliczyć pierwiastek z pierwszego operandu. Na przykład pierwiastek trzeciego stopnia z x wynosi $x^{**}(1.0/3.0)$. Podobnie $x^{**}y$ jest równe $z^{1/(x^{**}y)}$. Operator ** jest łączny prawostronnie, a więc $x^{**}y^{**}z$ jest równe $x^{**}(y^{**}z)$. Na koniec należy zauważyć, że operator ** ma wyższy priorytet niż jednoargumentowy operator -. Dlatego $-1^{**}0.5$ jest równe $-z^{(1^{**}0.5)}$. Aby obliczyć pierwiastek drugiego stopnia z -1 należy użyć nawiasów — $(-1)^{**}0.5$ (liczba urojona będąca wynikiem jest typu NaN).

4.6.3. Operatory arytmetyczne: +, -, *, / oraz %

Operatory +, -, * i / wykonują działania dodawania, odejmowania, mnożenia i dzielenia obiektów wszystkich klas pochodnych od klasy Numeric. Wynikiem dzielenia liczb całkowitych jest liczba całkowita po odrzuceniu reszty z dzielenia. Resztę można obliczyć za pomocą operatora modulo %. Dzielenie liczby całkowitej przez zero zwraca błąd ZeroDivisionError.

Dzielenie liczby zmiennoprzecinkowej przez zero zwraca dodatnią lub ujemną nieskończoność (Infinity). Dzielenie zmiennoprzecinkowe zera przez zero zwraca wartość NaN. Więcej informacji na temat działań arytmetycznych na liczbach całkowitych i zmiennoprzecinkowych znajduje się w podrozdziale 3.1.3.

W klasie `String` operator `+` wykonuje konkatenację, `*` powtarzałań, a `%` służy do zamiany argumentów metody `sprintf` nałańcuchy.

W klasie `Array` operator `+` jest używany do konkatenacji, a `-` do odejmowania tablic. Operator `*` działa na różne sposoby w zależności od klasy drugiego operandu. Kiedy tablica jest mnożona przez liczbę, wynikiem jest nowa tablica zawierająca zawartość oryginalnej tablicy powtórzoną określoną liczbę razy. Jeśli jednak tablica jest mnożona przezłańcuch, wynik jest taki sam jak przy wywołaniu metody `join` tej tablicy i przekazaniułańcucha jako argumentu.

4.6.4. Przesunięcie i dołączanie: `<< i >>`

W klasach `Fixnum` i `Bignum` operatory `<< i >>` przesuwają bity lewego argumentu w lewo i prawo. Argument po prawej stronie określa liczbę bitów przesunięcia. Ujemna liczba oznacza przesunięcie w drugą stronę — na przykład przesunięcie w lewo o `-2` jest równoważne z przesunięciem w prawo o `2`. Bitów wysokie nie są nigdy tracone przy przesuwaniu bitów w obiektach klasy `Fixnum` w lewo. Jeśli wynik przesunięcia nie mieści się w klasie `Fixnum`, zwarcana jest wartość klasy `Bignum`. Natomiast przesunięcia w prawo zawsze powodują utratę bitów niższych argumentu.

Przesunięcie liczby w lewo o jeden bit jest równoznaczne z jej pomnożeniem przez dwa. Przesunięcie w prawo o jeden bit odpowiada podzieleniu całkowitoliczbowemu przez dwa. Poniżej znajduje się kilka przykładów, w których liczby zostały zapisane w notacji binarnej, a następnie przekonwertowane z powrotem na format binarny:

```
(Ob1011 << 1).to_s(2)    # => "10110" 11 << 1 => 22.  
(Ob10110 >> 2).to_s(2)  # => "101"   22 >> 2 => 5.
```

Operator `<<` jest także używany do dołączania i w tej funkcji jest prawdopodobnie częściej stosowany. W ten sposób jest on zdefiniowany między innymi w klasach `String`, `Array`, `I0`, `Queue` czy `Logger`:

```
message = "witaj"          # Lańcuch.  
messages = []               # Pusta tablica.  
message << " świecie"     # Dołączanie dołańcucha.  
messages << message      # Dołączanie komunikatu do tablicy.  
STDOUT << message        # Wydruk komunikatu w standardowym strumieniu wyjściowym.
```

4.6.5. Dopełnienie, suma, przecięcie: `~, &, |` oraz `^`

W klasach `Fixnum` i `Bignum` operatory te wykonują operacje logiczne NOT, AND, OR oraz XOR. Operator `~` jest jednoargumentowy i ma wysoki priorytet, pozostałe z wymienionych mają średni priorytet.

Operator `~` zamienia każdy bit 0 liczby całkowitej będącej jego operandem na 1, a każdy bit 1 na 0, dając binarne dopełnienie jedynkowe liczby. `~x` jest równoważne z `-x-1` dla każdej liczby całkowitej `x`.

Operator & to bitowy dwuargumentowy operator AND. Bitы w wartości zwrotnej są ustawiane na 1 tylko wówczas, gdy odpowiadające sobie bitы w każdym z operandów są ustawione na 1. Na przykład:

```
(Ob1010 & Ob1100).to_s(2) #=> "1000".
```

Znak | to bitowy operator OR działający na dwóch liczbach całkowitych. Bitы w wartości zwrotnej są ustawiane na 1 tylko wówczas, gdy jeden z odpowiadających sobie bitów operandów był ustawiony na 1. Na przykład:

```
(Ob1010 | Ob1100).to_s(2) #=> "1110".
```

Znak ^ jest bitowym operatorem XOR (lub wykluczające) działającym na liczbach całkowitych. Bitы w wartości zwrotnej są ustawiane na 1 tylko wówczas, gdy jeden (ale nie oba) z pary odpowiadających sobie bitów w operandach jest równy 1. Na przykład:

```
(Ob1010 ^ Ob1100).to_s(2) #=> "110".
```

Inne klasy również używają tych operatorów. Zazwyczaj ich działanie jest zgodne z logicznymi operacjami AND, OR i NOT. Tablice wykorzystują operatory & i | do operacji przecięcia i sumy zbiorów. Operator & zastosowany na rzecz dwóch tablic zwraca nową tablicę zawierającą tylko te elementy, które znajdują się zarówno w jednej, jak i w drugiej tablicy. Operator | zwraca w takiej sytuacji nową tablicę zawierającą wszystkie elementy, które znajdują się w jednej lub drugiej tablicy. Szczegóły na ten temat i przykłady znajdują się w rozdziale 9.5.2.7.

W klasach TrueClass, FalseClass i NilClass operatory &, | oraz ^ (nie ma operatora ~) są zdefiniowane jako operatory logiczne. Należy jednak pamiętać, że taki sposób ich użycia jest rzadko zalecany. Do operacji logicznych przeznaczone są operatory && i || (opisane dalej w podrozdziale 4.6.8). Są one szybsze, ponieważ nie sprawdzają wartości swojego prawego operandu, jeśli jego wartość nie ma wpływu na wynik działania.

4.6.6. Porównywanie: <, <=, >, >= oraz <=>

Niektóre klasy porządkują swoje wartości w porządku naturalnym. Liczby są porządkowane według wielkości, łańcuchy alfabetycznie, a daty chronologicznie. Operatory mniejszości (<), mniejszy lub równy (<=), większy lub równy (>=) oraz większości (>) tworzą asercje dotyczące wzajemnej kolejności dwóch wartości. Zwracają wartość true, jeśli asercja jest prawdziwa, lub false w przeciwnym przypadku (zazwyczaj zgłasza wyjątek, gdy argumenty nie są porównywalne).

Klasy mogą definiować każdy z operatorów porównywania osobno. Jednak łatwiej (i częściej się to robi) jest zdefiniować jeden operator porównywania <=>. Jest on przeznaczony do celów ogólnych, a jego wartość zwrotna określa względną kolejność dwóch operandów. Jeśli operand z lewej strony jest mniejszy od operandu z prawej strony, <=> zwraca wartość -1. W sytuacji gdy operand z lewej strony jest większy, zwracana jest wartość +1. Gdy operandy są równe, wartością zwrotną jest 0. Jeżeli natomiast operandy nie dadzą się porównać, zwracana jest wartość nil⁴. Kiedy zdefiniowany jest operator <=>, klasa może zawierać moduł Comparable, który definiuje pozostałe operatory porównywania (w tym operator ==) w kategoriach operatora <=>.

⁴ Niektóre implementacje tego operatora mogą zwracać każdą wartość mniejszą od zera lub większą od zera zamiast -1 lub +1. Implementując operator <=>, należy dopilnować, aby zwracał wartości -1, 0 lub +1. Jednak używając go, należy sprawdzać wartości mniejsze lub większe od zera, a nie zakładać, że wynikiem jest zawsze -1, 0 lub +1.

Na specjalne wyróżnienie zasługuje klasa `Module`. Operatory porównywania w tej klasie określają powiązania podklas (klaśa `Module` jest nadklassą klasy `Class`). A < B ma wartość `true` dla klas A i B, jeśli A jest podklassą, czyli potomkiem klasy B. W tym przypadku „mniejszy niż” oznacza „bardziej wyspecjalizowany”, czyli „o węższym typie”. Dodatkowo warto zapamiętać, że znak < jest używany także przy deklarowaniu podklas (jak napisaliśmy w rozdziale 7.):

```
# Deklaracja klasy A jako podklasy klasy B.  
class A < B  
end
```

Operator > w klasie `Module` działa tak samo jak < po zamianie miejscami operandów. Ponadto operatory <= i >= zwracają wartość `true`, jeśli oba operandy są tej samej klasy. Najbardziej interesującą rzeczą w operatorach porównywania klasy `Module` jest to, że definiuje ona tylko częściowy porządek swoich wartości. Biorąc pod uwagę klasy `String` i `Numeric`, obie są podklasami klasy `Object` i żadna nie jest podklassą tej drugiej. W przypadku kiedy operandy nie są ze sobą powiązane, operatory porównywania zwracają wartość `nil` zamiast `true` lub `false`:

<code>String < Object</code>	<code># true: Klasa String jest bardziej wyspecjalizowana niż Object.</code>
<code>Object > Numeric</code>	<code># true: Klasa Object jest bardziej ogólna niż klasa Numeric.</code>
<code>Numeric < Integer</code>	<code># false: Klasa Numeric nie jest bardziej wyspecjalizowana od klasy Integer.</code>
<code>String < Numeric</code>	<code># nil: Klasy String i Numeric nie są powiązane ze sobą.</code>

Kiedy klasa definiuje porządek całkowity dla swoich wartości i `a < b` nie jest prawdą, to jest pewne, że `a >= b` jest prawdą. Jeśli jednak klasa definiuje tylko porządek częściowy, nie można polegać na tym założeniu.

4.6.7. Równość: ==, !=, =~, !~ oraz ===

Operator `==` to operator równości. Określa, czy dwie wartości są równe zgodnie z definicją równości operandu po lewej stronie. Operator `!=` jest odwrotnością operatora `==` — wywołuje `==` i zwraca przeciwną wartość. Operator `!=` można przedefiniować od Ruby 1.9, wcześniej nie było to możliwe. Więcej informacji na temat równości obiektów znajduje się w podrozdziale 3.8.5.

Operator `=~` służy do dopasowywania wzorców. W klasie `Object` zwraca on zawsze wartość `false`. W klasie `String` jego prawym argumentem powinno być wyrażenie regularne, natomiast w klasie `Regexp` — łańcuch. Oba te operatory zwracają wartość `nil`, jeśli łańcuch nie pasuje do wzorca. W przeciwnym przypadku operatory zwracają indeks całkowitoliczbowy, od którego zaczyna się dopasowanie (przypomnijmy, że w wyrażeniach logicznych wartość `nil` jest odpowiednikiem wartości `false`, a wartości całkowitoliczbowe odpowiadają wartości `true`).

Operator `!~` jest odwrotnością operatora `=~` — wywołuje `=~` i zwraca wartość `true`, jeśli operator `=~` zwrócił wartość `nil`, lub `false`, gdy operator `=~` zwrócił liczbę całkowitą. Operator `!~` można przedefiniować od wersji Ruby 1.9.

Operator `===` to operator równości szczególnej (ang. *case equality*). Jest używany przez instrukcję `case` (zobacz rozdział 5.). Jest on znacznie rzadziej używany w sposób jawnym niż operator `==`. Klasy `Range`, `Class` i `Regexp` definiują go jako rodzaj operatora przynależności lub dopasowywania wzorców. Pozostałe klasy dziedziczą definicję po klasie `Object`, która w zamian wywołuje operator `==` (zobacz podrozdział 3.8.5). Należy zauważać, że nie ma operatora `!==` — aby zanegować operator `==`, należy to zrobić we własnym zakresie.

4.6.8. Operatory logiczne: &&, ||, !, and, or, not

Operatory logiczne w języku Ruby są wbudowane i nie są oparte na metodach — nie można na przykład zdefiniować w klasie własnego operatora `&&`. Powodem tego jest fakt, że operatory logiczne mogą być używane z dowolnymi wartościami i ich działanie musi być spójne bez względu na rodzaj operandów. W języku Ruby dostępne są specjalne wartości `true` i `false`, ale nie ma typu `Boolean` (logicznego). Dla celów operatorów logicznych wartości `false` i `nil` są uważane za fałsz. Wszelkie inne wartości, wliczając `true`, `0`, `NaN`, `" "`, `[]` i `{}`, są uznawane za prawdę. Należy zwrócić uwagę na wyjątkowy operator `!`, który można przeddefiniować od wersji Ruby 1.9 (w Ruby 1.8 jest to niemożliwe). Można też zdefiniować metody o nazwach `and`, `or` i `not`, ale będą to zwykłe metody niemające wpływu na działanie operatorów o tych samych nazwach.

Innym powodem, dla którego operatory logiczne stanowią wbudowaną część języka Ruby, zamiast być możliwymi do przedefiniowania metodami, jest to, że operatory dwuargumentowe korzystają ze skróconego wartościowania. Jeśli wartość operacji można określić na podstawie tylko lewego operandu, ten znajdujący się z prawej strony jest ignorowany i nigdy nie oblicza się jego wartości. Jeżeli po prawej stronie znajduje się wyrażenie wywołujące skutki uboczne (jak przypisanie lub wywołanie metody ze skutkami ubocznymi), skutek uboczny może, ale nie musi mieć miejsca. Zależy to od wartości lewego operandu.

Operatorem logicznym AND jest `&&`. Zwraca wartość typu `true`, jeśli obydwa jego operandy mają wartość `true`. W przeciwnym przypadku zwraca wartość typu `false`. Należy zauważyć, że zwracane wartości są określone jako **typu** `true` i `false`, a nie jako `true` i `false`. Operator `&&` jest często używany w połączeniu z operatorami porównywania jak `==` i `<` w wyrażeniach podobnych do poniższego:

```
x == 0 && y > 1
```

Operatory porównywania i równości zwracają wartości `true` i `false`, a więc w tym przypadku operator `&&` działa na prawdziwych wartościach logicznych. Jednak nie zawsze tak jest. Operatora tego można także używać w następujący sposób:

```
x && y
```

W tym przypadku `x` i `y` mogą być czymkolwiek. Wartością tego wyrażenia jest albo wartość `x`, albo wartość `y`. Jeśli zarówno `x`, jak i `y` są typu `true`, wartością wyrażenia jest wartość `y`. Gdy `x` ma wartość typu `false`, wartością wyrażenia jest wartość `x`. W przeciwnym razie `y` musi mieć wartość typu `false`, a wartością wyrażenia jest wartość `y`.

Jak w rzeczywistości działa operator `&&`? Najpierw sprawdza wartość swojego lewego operandu. Jeżeli jest to `nil` lub `false`, zwraca tę wartość i pomija operand z prawej strony. W przeciwnym przypadku lewy operand ma wartość typu `true` i wartość operatora `&&` zależy od wartości prawego operandu. Wtedy operator `&&` sprawdza wartość swojego prawego operandu i zwraca jego wartość.

Możliwość pominięcia prawego operandu przez operator `&&` można wykorzystać podczas pisania kodu. Przeanalizuj poniższe wyrażenie:

```
x && print(x.to_s)
```

Powyzsza procedura drukuje wartość x jako łańcuch, ale tylko wtedy, gdy nie jest to nil ani false⁵.

Operator `||` zwraca wynik logicznej operacji OR (LUB). Zwraca wartość typu `true`, jeśli którykolwiek z jego operandów ma wartość typu `true`. Jeżeli obydwa operandy mają wartość typu `false`, zwraca wartość typu `false`. Podobnie jak `&&`, operator `||` ignoruje swój prawy operand, jeśli nie ma on wpływu na wartość całej operacji. Działanie operatora `||` jest następujące: najpierw sprawdza wartość swojego lewego operandu. Jeśli jest to jakakolwiek wartość inna niż `nil` lub `false`, zwraca ją. W przeciwnym przypadku sprawdza wartość swojego prawego operandu i zwraca tę wartość.

Operator `||` może występować w roli łącznika kilku wyrażeń porównujących:

```
x < 0 || y < 0 || z < 0    # Czy któraś ze współrzędnych ma ujemną wartość?
```

W tym wypadku operandami operatora `||` będą rzeczywiste wartości `true` i `false`, a wynikiem również będzie `true` albo `false`. Niemniej jednak zastosowanie operatora `||` nie ogranicza się tylko do wartości `true` i `false`. Jednym z idiomatycznych użyć `||` jest zwrócenie pierwszej wartości różnej od `nil` z szeregu alternatyw:

```
# Jeśli argument x ma wartość nil, następuje pobranie jego wartości z tablicy asocjacyjnej preferencji użytkownika  
# lub domyślnej wartości zapisanej w stałej.
```

```
x = x || preferences[:x] || Defaults::X
```

Operator `&&` ma wyższy priorytet od operatora `||`. Weźmy na przykład poniższe wyrażenie:

```
1 || 2 && nil    #=> 1.
```

Najpierw wykonywany jest operator `&&`, a wartość tego wyrażenia to `1`. Gdyby jako pierwszy został wykonany operator `||`, zostałaby zwrócona wartość `nil`:

```
(1 || 2) && nil    #=> nil.
```

Operator `!` wykonuje jednoargumentową operację logiczną NOT (NIE). Jeśli operand ma wartość `nil` lub `false`, operator `!` zwraca wartość `true`. W przeciwnym przypadku zwraca wartość `false`.

Operator `!` ma najwyższy priorytet. Oznacza to, że chcąc obliczyć odwrotność wyrażenia, które zawiera inne operatory, należy użyć nawiasów:

```
!(a && b)
```

Przez przypadek jedna z zasad logiki boolowskiej pozwala na zapisanie powyższego wyrażenia następująco:

```
!a || !b
```

Operatory `and`, `or` i `not` są niskopriorytetowymi wersjami operatorów `&&`, `||` i `!`. Jednym z powodów do używania tych wersji jest to, że ich nazwy są angielskimi słowami, dzięki czemu kod jest łatwiejszy do odczytania. Spróbuj na przykład przeczytać poniższy wiersz kodu:

```
if x > 0 and y > 0 and not defined? d then d = Math.sqrt(x*x + y*y) end
```

Innym powodem do używania tych alternatywnych wersji operatorów logicznych jest to, że mają niższy priorytet niż operatory przypisania. Oznacza to, że można napisać wyrażenie logiczne podobne do przedstawionego poniżej, które przypisuje wartości do zmiennych, aż napotka wartość typu `false`:

⁵ To, że wyrażenie można zapisać w ten sposób, nie oznacza, że należy tak robić. W rozdziale 5. wyjaśniamy, że takie wyrażenie lepiej zapisać następująco:

```
print(x.to_s) if x
```

```
if a = f(x) and b = f(y) and c = f(z) then d = g(a,b,c) end
```

Niniejsze wyrażenie nie zadziałałoby, gdyby zamiast operatora `and` użyto `&&`.

Należy zauważyc, że operatory `and` i `or` mają taki sam priorytet (`not` nieco wyższy). Ponieważ operatory `and` i `or` mają taki sam priorytet, a operatory `&&` i `||` mają różne priorytety, poniższe dwa wyrażenia zwracają różne wartości:

```
x || y && nil      # Najpierw jest wykonywany operator && => x.  
x or y and nil     # Wykonywane od lewej do prawej => nil.
```

4.6.9. Zakresy i przerzutniki: .. i ...

Operatory `..` i `...` zostały wprowadzone w podrozdziale 3.5, gdzie zostały opisane jako komponent składowi literałów zakresowych. Kiedy punkty początkowy i końcowy zakresu są liczbami całkowitymi, jak na przykład w zakresie `1..10`, interpreter podczas analizy składniowej tworzy literal obiektu klasy `Range`. Jeśli natomiast punkt początkowy lub końcowy jest czymkolwiek bardziej złożonym niż literal całkowitoliczbowy, np. `x..2*x`, określenie „literal obiektu klasy `Range`” nie jest trafną nazwą. Jest to wtedy wyrażenie tworzące zakres. Z tego zatem wynika, że `..` i `...` to operatory, a nie tylko komponenty składowi literałów zakresowych.

Operatory `..` i `...` nie są metodami, a więc nie można ich przedefiniowywać. Mają względnie niski priorytet, dzięki czemu zazwyczaj można ich używać bez nawiasów wokół lewych lub prawych operandów:

```
x+1..x*x
```

Wartością tych operatorów jest obiekt klasy `Range`. `x..y` jest równoznaczne z:

```
Range.new(x,y)
```

`x...y` jest równoznaczne z:

```
Range.new(x,y,true)
```

4.6.9.1. Przerzutniki logiczne

Operatory `..` i `...` użyte w instrukcjach warunkowych jak `if` lub w pętlach jak `while` (więcej informacji na temat instrukcji warunkowych i pętli zawiera rozdział 5.) nie tworzą obiektów klasy `Range`. W zamian zwracają specjalny rodzaj wyrażenia logicznego o nazwie **przerzutnik** (ang. *flip-flop*). Wartością wyrażenia przerzutnikowego, podobnie jak wyrażeń porównujących, jest `true` lub `false`. Wyjątkowość wyrażenia przerzutnikowego polega na tym, że jego wartość zależy od wartości wcześniejszych obliczeń. Oznacza to, że wyrażenie przerzutnikowe ma określony stan — musi pamiętać informacje o poprzednich obliczeniach. Ze względu na posiadanie stanu można się spodziewać, iż wyrażenie przerzutnikowe jest jakimś rodzajem obiektu. Tak jednak nie jest — jest to wyrażenie języka Ruby, a stan (pojedyncza wartość logiczna) jest przechowywany przez interpreter w wewnętrznej przetworzonej reprezentacji tego wyrażenia.

Dysponując tymi podstawowymi informacjami, przyjrzyj się przerzutnikowi w poniższym fragmencie kodu. Zauważ, że pierwszy operator `..` zwraca obiekt klasy `Range`. Drugi tworzy wyrażenie przerzutnikowe:

```
(1..10).each { |x| print x if x==3..x==5 }
```

Przerzutnik składa się z dwóch wyrażeń logicznych połączonych operatorem .. w kontekście instrukcji warunkowej lub pętli. Wartością przerzutnika jest false, chyba że i dopóki wartością wyrażenia po lewej stronie jest true. Kiedy wartość tego wyrażenia zmienia się na true, wyrażenie przechodzi w stan true. Wyrażenie pozostaje w tym stanie i kolejne obliczenia zwracają wartość true, dopóki wyrażenie po prawej stronie nie zwróci wartości true. Gdy to się stanie, przerzutnik wraca do stanu false. Kolejne obliczenia tego wyrażenia zwracają wartość false, dopóki wyrażenie po lewej stronie nie zwróci znowu wartości true.

W przedstawionym przykładzie kodu przerzutnik jest obliczany wielokrotnie dla wartości x od 1 do 10. Zaczyna od stanu false i pozostaje w nim dla wartości x równych 1 i 2. Kiedy $x==3$, przerzutnik zmienia stan na true. Zwraca wartości true, kiedy x ma wartości 4 i 5. Gdy jednak $x==5$, przerzutnik zmienia stan z powrotem na false i zwraca wartości false dla wszystkich pozostałych wartości x. W rezultacie kod ten zwraca wartość 345.

Zarówno operator .., jak i ... mogą być używane w przerzutnikach. Różnica polega na tym, że kiedy przerzutnik utworzony za pomocą operatora .. zmienia stan na true, zwraca wartość true, ale dodatkowo sprawdza wyrażenia po prawej stronie, aby sprawdzić, czy nie powinien zmienić swojego stanu z powrotem na false. Wersja z operatorem ... sprawdza wyrażenie po prawej stronie dopiero przy kolejnym obliczaniu. Spójrz na poniższe dwa wiersze kodu:

```
# Drukuje "3". Zmienia stan i wraca do pierwotnego stanu, kiedy x==3.  
(1..10).each { |x| print x if x==3..x>=3 }  
# Drukuje "34". Zmienia stan, kiedy x == 3, i wraca do pierwotnego, kiedy x==4.  
(1..10).each { |x| print x if x==3...x>=3 } # Drukuje "34".
```

Przerzutniki należą do najbardziej niejasnych własności języka Ruby i najlepiej ich unikać. Nie oznacza to jednak, że występują tylko w tym języku. Ruby odziedziczył przerzutniki po języku Perl, który z kolei dziedziczy je po uniwersalnych narzędziach przetwarzania tekstu sed i awk⁶. Pierwotnym przeznaczeniem przerzutników było dopasowywanie wierszy tekstu między wzorcami początkowym a końcowym. Nadal są one przydatne w tym zastosowaniu. Poniższy prosty program Ruby demonstruje przerzutnik. Wczytuje plik tekstowy linijka po linijce i drukuje wszystkie linijki, które zawierają tekst „TODO”. Kontynuuje drukowanie aż do napotkania pustego wiersza:

```
ARGF.each do |line|    # Dla każdej linii na wejściu lub wymienionych plików.  
  print line if line=~~/TODO/..line=~/^$/ # Drukowanie linii, kiedy przerzutnik jest w stanie true.  
end
```

Trudno jest dokładnie opisać działanie przerzutnika w sposób formalny. Łatwiej to zrozumieć, studując kod, który działa w taki właśnie sposób. Poniższa funkcja działa tak samo jak przerzutnik $x==3..x==5$. Warunki z lewej i prawej strony zostały zapisane w samej funkcji, a stan przerzutnika jest przechowywany w zmiennej globalnej:

```
$state = false          # Globalna zmienna przechowująca stan przerzutnika.  
def flipflop(x)        # Sprawdzenie wartości zmiennej x względem przerzutnika.  
  if !$state            # Jeśli zapisany stan to false,  
    result = (x == 3)    # wynikiem jest wartość lewego operandu.  
    if result           # Jeżeli wynik ten to true,  
      $state = !(x == 5) # to zapisany stan nie pochodzi od prawego operandu.  
    end  
    result              # Zwrot wyniku.  
  else                  # W przeciwnym przypadku, jeśli zapisany stan to true,
```

⁶ Operator .. tworzy przerzutnik w stylu awk, a ... w stylu sed.

```

$state = !(x == 5)      # zapisanie odwrotności prawego operandu
true                   # i zwrot wartości true bez sprawdzania wartości lewego operandu.
end
end

```

Mając zdefiniowaną powyższą funkcję przerzutnika, można napisać poniższy kod, który — podobnie jak wcześniejszy przykład — drukuje 345:

```
(1..10).each { |x| print x if flipflop(x) }
```

Poniższa funkcja symuluje działanie przerzutnika $x==3 \dots x>=3$:

```

$state2 = false
def flipflop2(x)
  if !$state2
    $state2 = (x == 3)
  else
    $state2 = !(x >= 3)
  end
end
# Sprawdzenie działania funkcji.
(1..10).each { |x| print x if x==3...x>=3 } # Drukuje "34".
(1..10).each { |x| print x if flipflop2(x) } # Drukuje "34".

```

4.6.10. Operator warunkowy ?:

Operator warunkowy ?: jest jedynym operatorem trójargumentowym w języku Ruby. Pierwszy operand powinien znajdować się przed znakiem zapytania, drugi pomiędzy znakiem zapytania a dwukropkiem, trzeci natomiast za dwukropkiem.

Operator ?: zawsze sprawdza wartość swojego pierwszego operandu. Jeśli jest to coś innego niż `false` lub `nil`, wartością całego wyrażenia jest wartość drugiego operandu. W przeciwnym przypadku, jeśli pierwszy operand ma wartość `false` lub `nil`, wartością całego wyrażenia jest wartość trzeciego operandu. W każdej z tych sytuacji wartość jednego z operandów nie jest obliczana (co ma znaczenie w przypadku efektów ubocznych jak przypisanie). Oto przykładowe użycie niniejszego operatora:

```
"Masz #{n} #{n==1 ? 'wiadomość' : 'wiadomości'}"
```

Jak widać, operator ?: spełnia rolę kompaktowo zapisanej instrukcji `if-then-else` (instrukcja warunkowa `if` została opisana w rozdziale 5.). Pierwszy operand to warunek, który jest sprawdzany — podobnie jak wyrażenie po słowie kluczowym `if`. Drugi operand odpowiada kodowi po słowie kluczowym `then`. Trzeci natomiast jest jak kod znajdujący się po słowie kluczowym `else`. Różnica pomiędzy operatorem ?: a instrukcją warunkową `if` jest taka, że druga instrukcja `if` umożliwia napisanie dowolnej ilości kodu w klauzulach `then` i `else`, natomiast operator ?: pozwala tylko na pojedyncze wyrażenia.

Operator ?: ma bardzo niski priorytet, co oznacza, że nawiasy wokół jego operandów nie są z reguły potrzebne. Konieczność ich stosowania pojawia się, gdy pierwszy operand zawiera operator `defined?` lub jeden z pozostałych operandów zawiera operację przypisania. Należy pamiętać, że w języku Ruby znakiem zapytania mogą kończyć się także nazwy metod. Jeśli pierwszy operand operatora ?: kończy się identyfikatorem, konieczne jest umieszczenie tego operandu w nawiasach lub oddzielenie go od znaku zapytania operatora spacją. Jeżeli nie zostanie to zrobione, interpreter potraktuje znak zapytania operatora jako część wcześniejszego identyfikatora. Na przykład:

```

x==3?y:z      # To jest dozwolone.
3==?y:z       # Błąd składni: 'x?' jest interpretowane jako nazwa metody.
(3==x)?y:z    # OK: nawiasy rozwiązuje problem.
3==x ?y:z     # Spacje również rozwiązuje problem.

```

Znak zapytania musi znaleźć się w tym samym wierszu co pierwszy argument. W Ruby 1.8 dwukropki musi być w tym samym wierszu co drugi argument. Natomiast w Ruby 1.9 przed dwukropkiem może znajdować się znak nowego wiersza. Wtedy jednak po dwukropku musi znajdować się spacja, aby uniknąć pomyłki z literałem symbolu.

Z tabeli 4.2 (przedstawionej wcześniej w tym rozdziale) wynika, że operator `?:` jest łączny prawostronnie. Jeśli operator ten jest użyty dwa razy w jednym wyrażeniu, ten po prawej stronie jest grupowany:

```

a ? b : c ? d : e      # To wyrażenie...
a ? b : (c ? d : e)    # jest obliczane w taki sposób.
(a ? b : c) ? d : e    # NIE w taki.

```

Tego typu dwuznaczność jest jednak rzadka w przypadku operatora `?:`. Poniższe wyrażenie sprawdza największą wartość z trzech zmiennych za pomocą trzech operatorów warunkowych. Nie są wymagane żadne nawiasy (wymagane są natomiast spacje przed znakami zapytania), ponieważ instrukcja ta może zostać zinterpretowana tylko w jeden sposób:

```

max = x>y ? x>z ? x : z : y>z ? y : z
max = x>y ? (x>z ? x : z) : (y>z ? y : z)  # Z nawiasami.

```

4.6.11. Operatory przypisania

O wyrażeniach przypisania była już mowa w podrozdziale 4.5. Warto w tym miejscu napisać kilka słów o operatorach przypisania, które w tych wyrażeniach są używane. Po pierwsze, wartością wyrażenia przypisania jest wartość (lub tablica wartości) znajdująca się po prawej stronie operatora przypisania. Po drugie, operatory przypisania są łączne prawostronnie. Dzięki wymienionym zasadom możliwe jest działanie poniższego wyrażenia:

```

x = y = z = 0      # Przypisanie zera do zmiennych x, y i z.
x = (y = (z = 0))  # Niniejsze wyrażenie pokazuje kolejność wykonywania działań.

```

Po trzecie, należy pamiętać, że przypisanie ma bardzo niski priorytet. Reguły priorytetowości stanowią, że prawie wszystko, co znajduje się za operatorem przypisania, zostanie obliczone przed nim. Głównymi wyjątkami od tej zasady są operatory `and`, `or` i `not`.

Na koniec warto zauważyć, że mimo iż operatorów przypisania nie można definiować jako metod, złożone operatory przypisania zawierają operatory przeddefiniowywalne jak na przykład `+`. Przeddefiniowanie operatora `+` nie ma wpływu na przypisanie wykonywane przez operator `+=`, ale wpływa na dodawanie wykonywane przez ten operator.

4.6.12. Operator defined?

Jednoargumentowy operator `defined?` sprawdza, czy jego operand jest zdefiniowany, czy nie. Normalnie użycie niezdefiniowanej zmiennej lub metody powoduje wyjątek. Jeśli w wyrażeniu po prawej stronie operatora `defined?` użyte są niezdefiniowane zmienne lub metody (wliczając operatory zdefiniowane jako metody), operator ten zwraca wartość `nil`. Podobnie operator `defined?` zwraca wartość `nil`, gdy jego operand jest wyrażeniem z instrukcją `yield`

lub metodą super użytą w niewłaściwym miejscu (to jest kiedy nie ma bloku, do którego należy zwrócić wartość, lub metody nadklasy, którą należy wywołać). Ważne jest, aby zrozumieć, że wyrażenie będące operandem operatora defined? nie jest obliczane — operator sprawdza tylko, czy **mogłoby** zostać obliczone bez spowodowania błędu. Oto typowy sposób użycia operatora defined?:

```
# Obliczenie f(x), ale tylko jeśli f i x są zdefiniowane.
y = f(x) if defined? f(x)
```

Jeżeli operand jest zdefiniowany, operator defined? zwraca łańcuch. Zawartość tego łańcucha zazwyczaj jest bez znaczenia. Ważne, że jest to wartość typu true — nie nil czy false. Jednak w wartości zwrotnej przez ten operator można znaleźć pewne informacje na temat typu wyrażenia znajdującego się po prawej stronie. Tabela 4.3 zawiera zestawienie wartości, które może zwrócić niniejszy operator.

Tabela 4.3. Wartości zwrotne operatora defined?

Typ wyrażenia w operandzie	Wartość zwrotna
Referencja do zdefiniowanej zmiennej lokalnej	"local-variable"
Referencja do zdefiniowanej zmiennej lokalnej blokowej (tylko Ruby 1.8)	"local-variable(in-block)"
Referencja do zdefiniowanej zmiennej globalnej	"global-variable"
Specjalne zmienne globalne wyrażeń regularnych: \$&, \$+, \$^, \$' oraz \$! do \$9, kiedy zdefiniowane po pomyślnym dopasowaniu (tylko Ruby 1.8)	Nazwa zmiennej w formie łańcucha
Referencja do zdefiniowanej stałej	"constant"
Referencja do zdefiniowanej zmiennej obiektowej	"instance-variable"
Referencja do zdefiniowanej zmiennej klasowej	"class variable" (zauważ brak myślnika)
nil	"nil" (to jest łańcuch)
true, false	"true", "false"
self	"self"
yield, kiedy nie ma bloku, do którego ma być przesłana wartość (zobacz też metodę block_given? z klasy Kernel)	"yield"
super w dozwolonym miejscu	"super"
Przypisanie (przypisanie nie jest w rzeczywistości wykonywane)	"assignment"
Wywołanie metody, wliczając operatory zdefiniowane jako metody (metoda nie jest w rzeczywistości wywoływana, a więc nie musi posiadać prawidłowej liczby argumentów — zobacz także metodę Object.respond_to?)	"method"
Każde inne poprawne wyrażenie, wliczając literaly i wbudowane operatory	"expression"
Każde wyrażenie, w którym użyto nazwy niezdefiniowanej zmiennej lub metody, albo instrukcji yield, lub metody super w niedozwolonych miejscach	nil

Operator defined? ma bardzo niski priorytet. Aby sprawdzić, czy dwie zmienne są zdefiniowane, należy użyć operatora and zamiast &:>

```
defined? a and defined? b      # To działa.
defined? a && defined? b      # Obliczane jako: defined?((a && defined? b)).
```

4.6.13. Modyfikatory instrukcji

Słowa kluczowe `rescue`, `if`, `unless`, `while` i `until` oznaczają instrukcje warunkowe, pętle i instrukcje obsługi wyjątków, które wpływają na przepływ sterowania w programie. Można ich także używać w roli modyfikatorów instrukcji. Na przykład:

```
print x if x
```

W tej formie można je uznać za operatory, w których wartość po prawej stronie ma wpływ na wykonanie wyrażenia po lewej stronie (lub w przypadku modyfikatora `rescue` stan wyjątku wyrażenia z lewej strony wpływa na wykonywanie operandu z prawej strony).

Opis tych słów kluczowych jako operatorów nie jest zbyt przydatny. Zostały one opisane zarówno w formie wyrażeniowej, jak i instrukcyjnej w rozdziale 5. Słowa te zostały wymienione w tabeli 4.2, aby pokazać ich miejsce wśród innych operatorów. Wszystkie mają bardzo niski priorytet, ale modyfikator `rescue` ma wyższy priorytet niż przypisanie.

4.6.14. Inne znaki

Większość operatorów języka Ruby jest zapisywana przy użyciu znaków interpunkcyjnych. W gramatyce tego języka są używane również znaki interpunkcyjne niebędące operatorami. Mimo iż większość z nich była już przedstawiona (lub będzie), tutaj zamieszczamy ich zestawienie:

- ()
Nawiasy są opcjonalnym składnikiem definicji metody i składni wywołania. Lepiej wywołanie metody traktować jako specjalny rodzaj wyrażenia, niż traktować znaki () jako operator wywołania metody. Nawiasy są także używane do grupowania podwyrażeń w celu zmiany kolejności wykonywania działań.
- []
Nawiasy kwadratowe są używane w literałach tablicowych, jak również do sprawdzania i ustawiania wartości w tablicach zwykłych i asocjacyjnych. W takim kontekście są ozdobnikiem syntaktycznym dla wywołań metod i działają podobnie do przeddefiniowalnych operatorów z dowolną krotnością. Zobacz podrozdziały 4.4 i 4.5.3.
- { }
Nawiasów klamrowych można używać zamiast ograniczników `do-end` w blokach oraz w literałach tablic asocjacyjnych.
- . i ::
. i :: są używane w nazwach kwalifikowanych. Oddzielają nazwę metody od obiektu, na rzecz którego metoda ta jest wywołana, lub nazwę stałej od modułu, w którym jest zdefiniowana. Nie są to operatory, ponieważ po prawej stronie nie ma wartości, tylko identyfikator.
- ; , , i =>
Niniejsze znaki interpunkcyjne są separatorami, a nie operatorami. Średnik (;) służy do oddzielania instrukcji wpisanych w jednym wierszu, przecinek (,) oddziela argumenty oraz elementy literałów tablic zwykłych i asocjacyjnych. Strzałka (=>) oddziela klucze od wartości w literałach tablic asocjacyjnych.

:

Dwukropki poprzedza literały symboli oraz jest używany w składni tablic asocjacyjnych w Ruby 1.9.

*, & i <

Niniejsze znaki interpunkcyjne w niektórych sytuacjach są operatorami, a w niektórych nie. Gwiazdka przed tablicą w instrukcji przypisania lub wywołaniu metody powoduje rozbicie tej tablicy na poszczególne elementy. Mimo iż czasami jest ona nazywana operatorem splat, w rzeczywistości nie jest operatorem — *a nie jest samodzielnym wyrażeniem.

Znaku & można używać w deklaracjach metod przed nazwą ich ostatniego argumentu, co powoduje, że jakikolwiek blok przekazany do metody zostaje przypisany do tego argumentu (zobacz rozdział 6.). Może też być używany w wywołaniach metod do przekazywania obiektów procedurowych, jakby były blokami.

Znak < służy do określania nadklasy w definicjach klas.

Instrukcje i przepływy sterowania



Spójrz na poniższy program. Dodaje on dwie liczby wprowadzone za pośrednictwem wiersza poleceń i drukuje ich sumę:

```
x = ARGV[0].to_f # Konwersja pierwszego argumentu na liczbę.  
y = ARGV[1].to_f # Konwersja drugiego argumentu na liczbę.  
sum = x + y      # Dodanie argumentów.  
puts sum         # Wydruk sumy.
```

Głównymi częściami tego prostego programu są przypisania wartości do zmiennych i wywołania metod. Swoją prostotę w szczególności zawdzięcza on czysto sekwencyjnemu sposobowi działania. Cztery tworzące go wiersze są wykonywane jeden po drugim bez żadnych rozwidleń czy powtórzeń. Tak proste programy są niezwykle rzadkie. Niniejszy podrozdział opisuje konstrukcje sterujące, które zmieniają taki sekwencyjny sposób wykonywania, czyli przepływ sterowania w programie. Opisane zostały:

- instrukcje warunkowe,
- pętle,
- iteratory i bloki,
- instrukcje zmieniające przepływ sterowania typu `return` i `break`,
- wyjątki,
- specjalne instrukcje `BEGIN` i `END`,
- osobliwe konstrukcje sterujące zwane wątkami i kontynuacjami.

5.1. Instrukcje warunkowe

Najczęściej używaną konstrukcją sterującą we wszystkich językach programowania jest instrukcja warunkowa. Pozwala ona na wykonywanie wyznaczonego bloku kodu tylko wówczas, gdy zostanie spełniony określony warunek. Warunek jest wyrażeniem — jeśli jego wartością nie jest `false` ani `nil`, oznacza to, że warunek został spełniony.

Język Ruby dysponuje szeroką gamą słów kluczowych tworzących instrukcje warunkowe. Dostępne opcje syntaktyczne zostały opisane w kolejnych podrozdziałach. Pisząc program w języku Ruby, można wybrać dowolną z nich, tę, która najlepiej nadaje się do konkretnego zastosowania.

5.1.1. Instrukcja warunkowa if

Najmniej skomplikowaną strukturą warunkową jest `if`. W swojej najprostszej formie wygląda następująco:

```
if wyrażenie  
  kod  
end
```

Kod znajdujący się pomiędzy słowami `if` a `end` jest wykonywany tylko wówczas, gdy **wyrażenie** ma wartość inną niż `false` lub `nil`. **Kod** musi być oddzielony od **wyrażenia** znakiem nowego wiersza, średnikiem lub słowem kluczowym `then`¹. Oto dwa sposoby na napisanie tej samej prostej instrukcji warunkowej:

¹ W Ruby 1.8 można także użyć dwukropka. W Ruby 1.9 jest to niedozwolone.

```

# Jeśli x jest mniejsze od 10, zwiększą jego wartość.
if x < 10                                # Znak nowego wiersza jako separator.
    x += 1
end
if x < 10 then x += 1 end      # Słowo kluczowe then jako separator.

```

Można także użyć jako separatora słowa kluczowego `then`, a po nim wstawić jeszcze znak nowego wiersza. Dzięki temu kod stanie się bardziej niezawodny i będzie działał nawet po usunięciu znaku nowego wiersza:

```

if x < 10 then
    x += 1
end

```

Programiści przyzwyczajeni do języka C lub jego pochodnych powinni zwrócić uwagę na dwie ważne rzeczy:

- Nawiasy wokół wyrażeń warunkowych nie są wymagane (i zazwyczaj nie są stosowane). Zamiast nich ogranicznikiem wyrażenia jest znak nowego wiersza, średnik lub słowo kluczowe `then`.
- Słowo kluczowe `end` jest wymagane, nawet jeśli wykonywany warunkowo kod składa się z jednej instrukcji. Modyfikatorowa forma `if` opisana niżej pozwala na pisanie prostych instrukcji warunkowych bez słowa `end`.

5.1.1.1. Klauzula `else`

Każda instrukcja warunkowa `if` może zawierać klauzulę `else` z kodem, który zostaje wykonany, kiedy warunek nie jest spełniony:

```

if wyrażenie
    kod
else
    kod
end

```

Kod znajdujący się pomiędzy słowami kluczowymi `if` a `else` jest wykonywany wówczas, gdy wyrażenie ma wartość inną niż `false` i `nil`. W przeciwnym przypadku, gdy wyrażenie ma wartość `false` lub `nil`, wykonywany jest kod między słowami `else` i `end`. Podobnie jak w najprostszej formie instrukcji `if`, wyrażenie musi być oddzielone od następującego po nim kodu znakiem nowego wiersza, średnikiem lub słowem kluczowym `then`. Druga część kodu jest w pełni oddzielona przez słowa kluczowe `else` i `end`, a więc w tym wypadku nie jest potrzebny znak nowego wiersza ani żadne inne ograniczniki.

Poniżej znajduje się przykładowa instrukcja warunkowa z klauzulą `else`:

```

if data           # Jeśli tablica ta istnieje,
    data << x   # dopisz do niej wartość.
else             # W przeciwnym przypadku...
    data = [x]   # utwórz nową tablicę dla tej wartości.
end              # Koniec instrukcji warunkowej.

```

5.1.1.2. Klauzula `elsif`

Aby sprawdzić więcej niż jeden warunek w instrukcji warunkowej, można pomiędzy słowami `if` i `else` wstawić dowolną liczbę klauzuł `elsif`. Nazwa `elsif` jest skrótem od „`else if`”. Zauważ, że w `elsif` jest tylko jedno „`e`”. Instrukcja warunkowa z użyciem klauzuli `elsif` wygląda następująco:

```

if wyrażenie1
    kod1
elsif wyrażenie2
    kod2
.
.
.
elsif wyrażenieN
    kodN
else
    kod
end

```

Jeśli wartością pierwszego wyrażenia jest cokolwiek innego niż `false` lub `nil`, wykonywany jest pierwszy blok kodu. W przeciwnym przypadku sprawdzana jest wartość drugiego wyrażenia. Jeśli nie jest to `false` ani `nil`, wykonywany jest drugi blok kodu. Proces ten jest kontynuowany do czasu, aż któreś z wyrażeń będzie miało wartość inną niż `false` lub `nil` albo skończą się klauzule `elsif`. Jeżeli wartością wyrażenia ostatniej klauzuli `elsif` jest `false` lub `nil`, a po niej znajduje się klauzula `else`, wykonywany jest kod zawarty między słowami `else` i `end`. W sytuacji gdy nie ma klauzuli `else`, nie jest wykonywany żaden blok kodu.

Klauzula `elsif` jest podobna do instrukcji `if` — wyrażenie musi być oddzielone od kodu znakiem nowego wiersza, średnikiem lub słowem kluczowym `then`. Oto przykład rozgałęzionej instrukcji warunkowej z wykorzystaniem klauzuli `elsif`:

```

if x == 1
  name = "jeden"
elsif x == 2
  name = "dwa"
elsif x == 3 then name = "trzy"
elsif x == 4; name = "cztery"
else
  name = "duzo"
end

```

5.1.1.3. Wartość zwrotna

W większości języków programowania `if` jest instrukcją warunkową. Jednak w języku Ruby wszystko jest wyrażeniem, nawet struktury sterujące, które powszechnie nazywane są instrukcjami. Wartością zwrotną „instrukcji” `if` (tj. wartością powstałą w wyniku obliczenia wartości wyrażenia `if`) jest wartość ostatniego wyrażenia, które zostało obliczone, lub `nil`, jeśli nie został wykonany żaden blok kodu.

Dzięki temu że instrukcje `if` zwracają wartości, zaprezentowaną wcześniej wielostopniową instrukcję warunkową można zapisać bardziej elegancko w następujący sposób:

```

name = if x == 1 then "jeden"
        elsif x == 2 then "dwa"
        elsif x == 3 then "trzy"
        elsif x == 4 then "cztery"
        else              "duzo"
end

```

5.1.2. Słowo kluczowe `if` jako modyfikator

Jeśli słowo `if` jest używane w swojej normalnej formie wyrażeniowej, gramatyka Ruby wymaga, aby koniec tego wyrażenia wyznaczało słowo kluczowe `end`. Reguła ta jest mało elegancka w przypadku prostych jednowierszowych instrukcji warunkowych. Problem ten ma

związek z analizą składniową, a rozwiązać go można przy użyciu słowa kluczowego `if` jako separatora kodu do wykonania od wyrażenia warunkowego. Zamiast pisać:

```
if wyrażenie then kod end
```

można napisać prościej:

```
kod if wyrażenie
```

W tej formie słowo kluczowe `if` jest **modyfikatorem** instrukcji (lub wyrażenia). Programiści Perla powinni być przyzwyczajeni do takiej składni. Pozostali muszą zapamiętać, że kod, który ma być wykonany, występuje pierwszy, a wyrażenie znajduje się za słowem kluczowym `if`. Na przykład:

```
puts message if message      # Wysłanie na wyjście komunikatu, jeśli został zdefiniowany.
```

Składnia ta kładzie większy nacisk na kod, który ma zostać wykonany, a mniejszy na warunek, który musi zostać spełniony, aby wykonanie kodu mogło nastąpić. Składnia ta może poprawić czytelność kodu, jeśli warunek jest bardzo prosty lub prawie zawsze ma wartość `true`.

Mimo iż warunek znajduje się na końcu, jego wartość jest sprawdzana na początku. Jeśli wartością nie jest `false` ani `nil`, następuje wykonanie kodu i wartością zwrotną całego zmodyfikowanego wyrażenia zostaje wartość zwrócona przez ten kod. W przeciwnym przypadku kod nie jest wykonywany, a wartością wyrażenia jest `nil`. Oczywiście w tej składni nie można używać klauzuli `else`.

Przed `if` użytym w roli modyfikatora musi bezpośrednio znajdować się modyfikowana instrukcja lub modyfikowane wyrażenie bez żadnych znaków nowego wiersza w środku. Wstawienie znaku nowego wiersza do poprzedniego przykładu zamieniło go w niezmodyfikowane wywołanie metody z niekompletną instrukcją `if`:

```
puts message          # Wyrażenie bezwarunkowe.  
if message           # Niekompletne!
```

Modyfikator `if` ma bardzo niski priorytet i wiąże znacznie słabiej niż operator przypisania. Należy mieć pewność, jakie wyrażenie jest za jego pomocą modyfikowane. Na przykład dwa poniższe wiersze kodu są różne:

```
y = x.invert if x.respond_to? :invert  
y = (x.invert if x.respond_to? :invert)
```

W pierwszym wierszu modyfikator jest stosowany do wyrażenia przypisania. Jeśli `x` nie udostępnia metody o nazwie `invert`, nic się nie dzieje i wartość `y` nie zostanie zmodyfikowana. W drugim wierszu modyfikator `if` jest zastosowany tylko do wywołania metody. Jeśli `x` nie udostępnia metody `invert`, wartością zmodyfikowanego wyrażenia jest `nil`, która zostanie przypisana do `y`.

Modyfikator `if` wiąże się z najbliższym pojedynczym wyrażeniem. Aby zmodyfikować więcej niż jedno wyrażenie, należy użyć nawiasów lub instrukcji `begin`. Z tym jest jednak problem, ponieważ osoba czytająca taki kod nie wie, że ma do czynienia z konstrukcją warunkową, dopóki nie dojdzie do samego jej dołu. Ponadto użycie modyfikatora `if` w ten sposób gubi zwięzłość, która jest główną zaletą tej składni. Kiedy w grę wchodzi więcej niż jeden wiersz kodu, lepiej użyć zwykłej instrukcji `if`, zamiast stosować `if` jako modyfikator. Oto porównanie trzech alternatywnych rozwiązań:

```
if wyrażenie      begin      (  
  wiersz1        wiersz1      wiersz1  
  wiersz2        wiersz2      wiersz2  
end             end if wyrażenie    ) end if wyrażenie
```

Warto zauważyć, że wyrażenie zmodyfikowane przez `if` samo w sobie jest wyrażeniem, która można zmodyfikować. W związku z tym możliwe jest dodanie do niego kilku modyfikatorów `if`:

```
# Wyślanie komunikatu na wyjście, jeśli istnieje i jeśli metoda wysyłająca jest zdefiniowana.  
puts message if message if defined? puts
```

Takie powtórzenia modyfikatora `if` są jednak trudne do odczytu i lepiej połączyć dwie konstrukcje warunkowe w jednym wyrażeniu:

```
puts message if message and defined? puts
```

5.1.3. Słowo kluczowe unless

Słowo kluczowe `unless` używane jako instrukcja lub modyfikator jest przeciwnieństwem słowa `if` — wykonuje kod tylko wówczas, gdy związane z nim wyrażenie ma wartość `false` lub `nil`. Jego składnia jest taka sama jak `if`, z tym wyjątkiem, że nie można używać klauzul `elsif`:

```
# Jednościeżkowa instrukcja unless.  
unless warunek  
  kod  
end  
# Dwuścieżkowa instrukcja unless.  
unless warunek  
  kod  
else  
  kod  
end  
# Modyfikator unless.  
kod unless warunek
```

W instrukcji `unless`, podobnie jak w `if`, wymagane jest, aby warunek i kod były rozdzielone znakiem nowego wiersza, średnikiem lub słowem kluczowym `then`. Dodatkowo instrukcje `unless`, podobnie jak `if`, są wyrażeniami i zwracają wartość wykonanego przez siebie kodu lub wartość `nil`, jeśli nic nie wykonują:

```
# Wywołanie metody to_s na rzecz obiektu o, chyba że o ma wartość nil.  
s = unless o.nil?  
  o.to_s  
end  
s = unless o.nil? then o.to_s end # Słowo kluczowe then jako separator.
```

W przypadku jednowierszowych konstrukcji warunkowych jak ta, bardziej przejrzysta jest zazwyczaj modyfikatorowa forma `unless`:

```
s = o.to_s unless o.nil?
```

W języku Ruby nie ma odpowiednika klauzuli `elsif` dla instrukcji warunkowych `unless`. Jeśli jednak komuś nie przeszkadza nieco bardziej rozwlekły styl, istnieje możliwość tworzenia wielościeżkowych instrukcji `unless`:

```
unless x == 0  
  puts "x nie jest 0"  
else  
  unless y == 0  
    puts "y nie jest 0"  
  else  
    unless z == 0  
      puts "z nie jest 0"  
    else
```

```

    puts "wszystkie są 0"
end
end
end

```

5.1.4. Instrukcja case

Instrukcja `case` jest wielościeżkową instrukcją warunkową. Występuje ona w dwóch formach. Jej prosta (i rzadziej używana) forma jest tylko alternatywną składnią dla instrukcji `if-elsif-else`. Dwa poniższe ustawione jedno obok drugiego wyrażenia są równoważne:

<pre> name = case when x == 1 then "jeden" when x == 2 then "dwa" when x == 3 then "trzy" when x == 4 then "cztery" else "dużo" end </pre>	<pre> name = if x == 1 then "jeden" elsif x == 2 then "dwa" elsif x == 3 then "trzy" elsif x == 4 then "cztery" else "dużo" end </pre>
--	--

Jak widać w powyższym kodzie, instrukcja `case` zwraca wartość, tak jak instrukcja `if`. Podobnie jak w instrukcji `if`, słowo kluczowe `then` po klauzuli `when` można zastąpić znakiem nowego wiersza lub średnikiem²:

```

case
when x == 1
  "jeden"
when x == 2
  "dwa"
when x == 3
  "trzy"
end

```

Instrukcja `case` sprawdza każde z wyrażeń `when` w takiej kolejności, w jakiej zostały napisane. Jeśli znajdzie takie, które zwraca wartość `true`, oblicza instrukcje znajdujące się pomiędzy nim a kolejnym słowem `when` albo znajdującym się niżej słowem `else` lub `end`. Wartością całego wyrażenia `case` jest wartość ostatniej obliczonej instrukcji. Po znalezieniu klauzuli `when` zwracającej wartość `true` pozostałe klauzule `when` są ignorowane.

Klauzula `else` w instrukcji `case` nie jest obowiązkowa, ale jeśli już zostanie użyta, musi znajdować się na samym końcu, za wszystkimi klauzulami `when`. Jeśli żadna z klauzul `when` nie zwraca wartości `true` i istnieje klauzula `else`, wykonywany jest kod znajdujący się między słowami `else` a `end`. Wartością całej instrukcji `case` jest wartość ostatniego wyrażenia w tym bloku kodu. Gdy żadna klauzula `when` nie zwraca wartości `true` i nie ma klauzuli `else`, nie jest wykonywany żaden blok kodu, a wartością całej instrukcji `case` jest `nil`.

Z klauzulą `when` znajdującą się w obrębie instrukcji `case` może być związane więcej niż jedno wyrażenie (poszczególne wyrażenia są oddzielane przecinkami). Jeżeli którekolwiek z tych wyrażeń ma wartość `true`, wykonywany jest kod związanego z klauzulą `when`. W takiej prostej formie instrukcji `case` przecinki nie są szczególnie przydatne i działają jak operator `||`:

```

case
when x == 1, y == 0 then "x jest jeden lub y jest zero" # Niejasna składnia.
when x == 2 || y == 1 then "x jest dwa lub y jest jeden" # Łatwiejsze do zrozumienia.
end

```

² W Ruby 1.8 zamiast słowa kluczowego `then` można także używać dwukropka, podobnie jak w instrukcji `if`. Nie jest to jednak dozwolone w Ruby 1.9.

Wszystkie zaprezentowane do tej pory przykłady użycia instrukcji `case` demonstrują jej prostszą formę. Instrukcja ta ma jednak znacznie większe możliwości. Zauważ, że w większości przypadków po lewej stronie każdej klauzuli `when` znajduje się takie samo wyrażenie. W prostszej formie instrukcji `case` to powtarzalne wyrażenie z lewej strony klauzuli `when` zostało przeniesione do samej instrukcji `case`:

```
name = case x
  when 1                      # Tylko wartość do porównania z x.
    "jeden"
  when 2 then "dwa"           # Słowo kluczowe then zamiast znaku nowego wiersza.
  when 3; "trzy"              # Średnik zamiast znaku nowego wiersza.
  else "dużo"                # Opcjonalna klauzula else na końcu.
end
```

W tej formie instrukcji `case` wyrażenie związane z `case` jest obliczane tylko jeden raz, a następnie jest porównywane z wartościami zwracanymi przez wyrażenie `when`. Porównywania odbywają się w takiej kolejności, w jakiej zostały zapisane klauzule `when`. Wykonywany jest kod związany z pierwszą dopasowaną klauzulą `when`. Jeśli żadna klauzula `when` nie zostanie dopasowana, zostanie wykonany kod związany z klauzulą `else` (jeżeli istnieje). Wartość zwrotna takiej instrukcji `case` jest taka sama jak wartość zwrotna prostej formy `case` — wartość ostatniego obliczonego wyrażenia lub `nil` w przypadku braku pasującej klauzuli `when` lub `else`.

W instrukcji `case` ważne jest zrozumienie sposobu porównywania klauzul `when` z wyrażeniem znajdującym się za słowem kluczowym `case`. Porównywanie to jest wykonywane przez operator `==`. Jest on wywoływany na rzecz wartości wyrażenia `when`, przekazywana jest do niego wartość wyrażenia `case`. Dlatego powyższa instrukcja `case` jest równoważna z poniższą (z tym wyjątkiem, że powyżej wartość `x` jest obliczana tylko jeden raz):

```
name = case
  when 1 === x then "jeden"
  when 2 === x then "dwa"
  when 3 === x then "trzy"
  else "dużo"
end
```

Operator `==` jest **operatorem równości w instrukcjach `case`** (ang. *case equality operator*). W wielu klasach, jak na przykład `Fixnum`, `==` działa tak samo jak operator `==`. Niemniej jednak w niektórych klasach operator ten posiada interesujące implementacje. W klasie `Class` operator `==` sprawdza, czy operand z prawej strony jest egzemplarzem klasy, której nazwa znajduje się po lewej stronie. W klasie `Range` określa, czy wartość po prawej stronie mieści się w przedziale z lewej strony. W klasie `Regexp` sprawdza, czy tekst po prawej stronie pasuje do wzorca z lewej strony. W Ruby 1.9 operator `==` w klasie `Symbol` porównuje symbole i łańcuchy. Znając powyższe definicje, można napisać ciekawe instrukcje `case`, jak poniższe:

```
# Podejmowanie różnych działań w zależności od klasy obiektu x.
puts case x
  when String then "łańcuch"
  when Numeric then "liczba"
  when TrueClass, FalseClass then "wartość logiczna"
  else "inna"
end

# Obliczanie podatku dochodowego przy użyciu instrukcji case i obiektów klasy Range.
tax = case income
  when 0..7550
    income * 0.1
  when 7550..30650
    755 + (income-7550)*0.15
  when 30650..74200
    4220 + (income-30655)*0.25
```

```

when 74200..154800
  15107.5 + (income-74201)*0.28
when 154800..336550
  37675.5 + (income-154800)*0.33
else
  97653 + (income-336550)*0.35
end
# Pobranie danych od użytkownika i przetworzenie ich. Komentarze są ignorowane, a wyjście
# następuje po wpisaniu przez użytkownika słowa koniec.
while line=gets.chomp do # Zapytanie o dane przy każdej iteracji pętli.
  case line
  when /^$/#
    next
  when /^koniec$/i
    break
  else
    puts line.reverse
  end
end

```

Z klauzulą when może być związane więcej niż jedno wyrażenie. Poszczególne wyrażenia są oddzielane przecinkami, a operator === jest wywoływany na rzecz każdego z nich. Oznacza to, że jeden blok kodu można uruchomić za pomocą więcej niż jednej wartości:

```

def hasValue?(x)          # Definicja metody o nazwie hasValue?.
  case x
  when nil, [], "", 0    # Wielościeżkowa instrukcja warunkowa oparta na wartości x.
    false                # Jeśli nil==x || []==x || ""==x || 0==x, to
  else                  # wartością zwrótną metody jest false.
    true                 # W przeciwnym przypadku
  end                   # wartością zwrótną metody jest true.
end

```

Instrukcja case a instrukcja switch

Programiści Javy i innych języków wywodzących się od C znają instrukcję warunkową switch, która jest podobna do instrukcji case w Ruby. Jest jednak między nimi kilka ważnych różnic:

- W Javie i podobnych do niej językach instrukcja ta nosi nazwę switch, a jej klauzule mają nazwy case i default. W Ruby nazwa instrukcji to case, a klauzule nazwują się when i else.
- Instrukcja switch przekazuje sterowanie na początek odpowiedniej klauzuli. Od tego miejsca wykonywanie jest kontynuowane i może przejść w dół przez wszystkie klauzule aż do napotkania końca instrukcji switch lub instrukcji break albo return. Dzięki temu kilka klauzul case może odwoływać się do tego samego bloku kodu. W Ruby ten sam efekt jest osiągany poprzez wpisanie kilku wyrażeń oddzielonych przecinkami dla każdej klauzuli when. Instrukcja case w języku Ruby nie pozwala na przechodzenie w dół przez klauzule.
- W Javie i większości kompliwanych języków z podobną składnią do C wyrażenia związane z każdą etykietą case muszą być stałymi czasu komplikacji, a nie dowolnymi wyrażeniami czasu wykonywania. Dzięki temu kompilator może zaimplementować instrukcję switch za pomocą bardzo szybkiej tablicy przeglądowej. Ograniczenia tego pozbawiona jest instrukcja case w języku Ruby, a jej szybkość jest porównywalna z instrukcją if z kilkoma klauzulami elsif.

5.1.5. Operator ?:

Operator warunkowy ?: (opisany w podrozdziale 4.6.10) działa bardzo podobnie do instrukcji if, gdzie znak ? zastępuje słowo kluczowe then, a : — else. Pozwala on na pisanie zwięzłych instrukcji warunkowych:

```
def how_many_messages(n) # Obsługa liczby pojedynczej i mnogiej.  
  "Masz " + n.to_s + (n==1 ? " wiadomość." : " wiadomości.")  
end
```

5.2. Pętle

Niniejszy podrozdział opisuje instrukcje pętlowe while, until oraz for. Język Ruby pozwala także na definiowanie niestandardowych konstrukcji pętlowych zwanych **iteratorami**. Iteratory (zobacz podrozdział 5.3) są prawdopodobnie częściej używane niż standardowe pętle. Opis iteratorów znajduje się w dalszej części niniejszego rozdziału.

5.2.1. Pętle while i until

Podstawowe pętle języka Ruby to while i until. Pierwsza z nich wykonuje blok kodu tak długo, jak długo (while) spełniany jest określany warunek. Druga natomiast działa do momentu (until), gdy warunek zostanie spełniony. Na przykład:

```
x = 10          # Inicjacja licznika pętli.  
while x >= 0 do  
  puts x        # Powtarzanie, dopóki x jest większe lub równe 0.  
  x = x - 1    # Wydruk wartości x.  
end            # Odjęcie 1 od x.  
# Odliczanie do 10 za pomocą pętli until.  
x = 0          # Rozpoczęcie od wartości 0 (zamiast -1).  
until x > 10 do  
  puts x        # Powtarzanie, aż x będzie większe niż 10.  
  x = x + 1    # Koniec pętli.
```

Warunkiem pętli jest wyrażenie logiczne umieszczone między słowami kluczowymi while i do lub until i do. Ciało pętli jest kod Ruby znajdujący się między słowami do i end. Pętla while sprawdza swój warunek. Jeśli jego wartością nie jest false ani nil, wykonuje kod w swoim ciele, a następnie ponownie sprawdza wartość warunku. W ten sposób ciało jest wykonywane zero lub więcej razy, dopóki warunek jest spełniany (lub mówiąc ściślej, nie ma wartości false ani nil).

Pętla until ma działanie przeciwnie. Warunek jest sprawdzany, a ciało wykonywane, jeśli wartością warunku jest false lub nil. Oznacza to, że ciało pętli jest wykonywane zero lub więcej razy, dopóki warunek ma wartość false lub nil. Należy zauważać, że każdą pętlę while można przerobić na until, negując jej warunek. Wielu programistów zna pętlę while, a nigdy nie używało pętli until. Dlatego lepiej może używać while, chyba że konstrukcja until znacząco wpływa na poprawę czytelności kodu.

Słowo kluczowe do w pętli while lub until jest podobne do then w instrukcji if — może zostać zastąpione znakiem nowego wiersza lub średnikiem³.

³ W Ruby 1.8 zamiast słowa kluczowego do można także użyć dwukropka. W Ruby 1.9 nie jest to dozwolone.

5.2.2. Słowa kluczowe while i until jako modyfikatory

Jeśli ciało pętli stanowi tylko jedno wyrażenie, pętlę tę można zapisać w bardzo zwięzły sposób, używając za wyrażeniem słów kluczowych `while` i `until` jako modyfikatorów. Na przykład:

```
x = 0          # Inicjacja zmiennej pętlowej.  
puts x = x + 1 while x < 10    # Wysłanie na wyjście i inkrementacja w jednym wyrażeniu.
```

W tej składni słowo kluczowe `while` samo jest separatorem oddzielającym ciało pętli od jej warunku, nie są potrzebne słowa kluczowe `do` (lub znak nowego wiersza) i `end`. Warto porównać ten kod z tradycyjną pętlą `while` zapisaną w jednym wierszu:

```
x = 0  
while x < 10 do puts x = x + 1 end
```

Słowo kluczowe `until` może być używane jako modyfikator, tak samo jak `while`:

```
a = [1,2,3]          # Inicjacja tablicy.  
puts a.pop until a.empty?    # Usuwanie elementów z tablicy, aż będzie pusta.
```

Należy pamiętać, że słowa `while` i `until` użyte jako modyfikatory muszą znajdować się w tym samym wierszu co ciało, które modyfikują. Jeśli pomiędzy ciałem pętli a słowem kluczowym `while` lub `until` znajdzie się znak nowego wiersza, interpreter potraktuje to ciało jako nieniemydifikowane wyrażenie, a słowa `while` i `until` jako początek zwykłej pętli.

Kiedy słowa kluczowe `while` i `until` zostaną użyte jako modyfikatory pojedynczego wyrażenia, warunek pętli jest sprawdzany na samym początku, mimo że znajduje się za ciałem pętli wykonywanym zero lub więcej razy, tak samo jak w przypadku zwykłej pętli `while` i `until`.

Od tej reguły jest jeden specjalny wyjątek. Jeśli obliczane wyrażenie jest wyrażeniem złożonym ograniczonym słowami kluczowymi `begin` i `end`, ciało jest wykonywane najpierw, przed sprawdzeniem warunku:

```
x = 10          # Inicjacja zmiennej pętlowej.  
begin          # Początek wyrażenia złożonego: wykonywane co najmniej jeden raz.  
  puts x        # Drukuje x.  
  x = x - 1    # Zmniejsza x.  
end until x == 0 # Koniec wyrażenia złożonego i jego modyfikacja za pomocą pętli.
```

W rezultacie powstaje struktura przypominająca pętlę `do-while` z języków C, C++ i Java. Mimo podobieństwa do pętli `do-while` z innych języków ten specjalny przypadek z instrukcją `begin` jest niezbyt jasny i odradza się jego używanie. W przyszłych wersjach języka Ruby używanie modyfikatorów `while` i `until` ze słowami kluczowymi `begin` i `end` może zostać zabronione.

Warto wiedzieć, że jeśli modyfikator `until` zostanie zastosowany do grupy instrukcji zamkniętej w nawiasach, będzie działał w normalny sposób:

```
x = 0          # Inicjacja zmiennej pętlowej.  
(          # Początek wyrażenia złożonego — może być wykonane zero razy.  
  puts x        # Drukuje x.  
  x = x - 1    # Zmniejsza x.  
) until x == 0 # Koniec wyrażenia złożonego i jego modyfikacja za pomocą pętli.
```

5.2.3. Pętla for-in

Pętla `for` lub `for-in` iteruje przez elementy przeliczalnych obiektów (jak tablice). Przy każdej iteracji przypisuje jeden element do określonej zmiennej pętlowej, a następnie wykonuje ciało. Pętla `for` wygląda następująco:

```
for zmienna in kolekcja do
    ciało
end
```

W miejscu zmiennej może znaleźć się pojedyncza zmienna lub lista zmiennych oddzielonych przecinkami. Kolekcją może być każdy obiekt udostępniający metodę iteracyjną `each`. Do wielu obiektów języka Ruby, które udostępniają tę metodę, należą obiekty klas `Array` i `Hash`. Pętla `for-in` wywołuje metodę `each` na rzecz określonego obiektu. Kiedy iterator ten zwraca wartości, pętla `for` przypisuje każdą z nich (lub każdy zbiór wartości) do specjalnej zmiennej (lub specjalnych zmiennych), a następnie wykonuje kod znajdujący się w ciele. Podobnie jak w przypadku pętli `while` i `until`, słowo kluczowe `do` jest opcjonalne. Można je zastąpić znakiem nowego wiersza lub średnikiem.

Oto kilka przykładowych pętli `for`:

```
# Drukuję elementy tablicy.
array = [1,2,3,4,5]
for element in array
    puts element
end
# Drukuję klucze i wartości tablicy asocjacyjnej.
hash = { :a=>1, :b=>2, :c=>3 }
for key,value in hash
    puts "#{key} => #{value}"
end
```

Zmienna pętlowa lub zmienne pętli `for` nie mają zasięgu lokalnego ograniczonego do swojej pętli. Ich definicje pozostają nawet po wyjściu z pętli. Podobnie nowe zmienne zdefiniowane w ciele pętli istnieją także po zakończeniu jej działania.

To, że działanie pętli `for` jest uzależnione od metody iteracyjnej `each`, wskazuje, że pętle `for` działają w dużym stopniu podobnie do iteratorów. Na przykład zaprezentowana powyżej pętla `for` iterująca przez klucze i wartości tablicy asocjacyjnej może zostać również zapisana przy jawnym użyciu iteratora `each`:

```
hash = { :a=>1, :b=>2, :c=>3 }
hash.each do |key,value|
    puts "#{key} => #{value}"
end
```

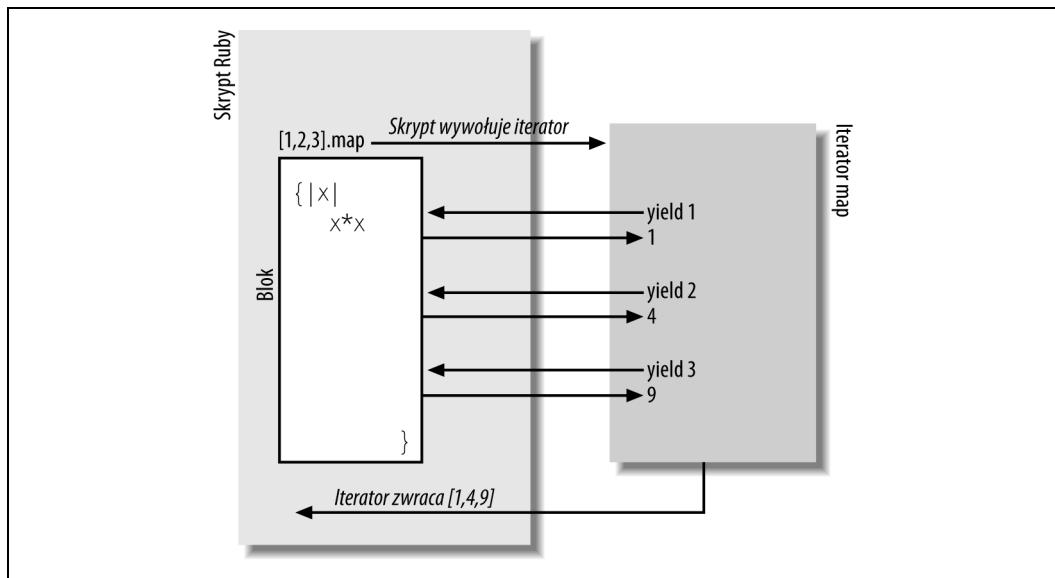
Jedyna różnica pomiędzy wersją `for` a wersją `each` polega na tym, że blok kodu znajdujący się za iteratorem definiuje nowy zakres dla zmiennych. Szczegółowe informacje na ten temat zostały przedstawione w dalszej części niniejszego rozdziału przy opisie iteratorów.

5.3. Iteratory i obiekty przeliczalne

Mimo iż pętle `while`, `until` i `for` są rdzennym składnikiem języka Ruby, prawdopodobnie częściej używa się zamiast nich tak zwanych metod iteracyjnych (iteratorów). Iteratory należą do najbardziej godnych uwagi własności języka Ruby. Przykłady podobne do poniższych często spotyka się w różnych wstępnych kursach do tego języka:

```
3.times { puts "dziękuję!" }      # Podziękowanie trzy razy.
data.each { |x| puts x }          # Wydruk każdego elementu x zmiennej data.
[1,2,3].map { |x| x*x }          # Obliczenie kwadratów elementów tablicy.
factorial = 1                     # Obliczenie silni n.
2.upto(n) { |x| factorial *= x }
```

Metody `times`, `each`, `map` i `upto` są iteratorami działającymi na **bloku** kodu, który znajduje się za nimi. Złożoną strukturą sterującą stojącą za tymi działania jest `yield`. Instrukcja `yield` zwraca tymczasowo sterowanie od iteratora do metody, która go wywołała. Sterowanie przepływa od iteratora do bloku kodu związanego z wywołaniem tego iteratora. Kiedy zostaje osiągnięty koniec bloku, iterator odzyskuje kontrolę i ponawia wykonywanie od pierwszego wiersza za instrukcją `yield`. Implementacja pętli zazwyczaj polega na wielokrotnym wywołaniu przez metodę iteracyjną tej instrukcji. Rysunek 5.1 przedstawia ten złożony przepływ sterowania. Bloki i instrukcja `yield` zostały szczegółowo opisane w podrozdziale 5.4. Teraz koncentrujemy się na samej iteracji, a nie na strukturze sterującej, która ją umożliwia.



Rysunek 5.1. Iterator przekazujący sterowanie do wywołującej go metody

Jak widać w przedstawionych wcześniej przykładach, bloki mogą być parametryzowane. Pionowe kreski na początku bloku pełnią podobną rolę do nawiasów w definicjach metod — przechowują listy nazw parametrów. Instrukcja `yield` jest jak wywołanie metody. Może znajdować się po niej zero lub więcej wyrażeń, których wartości są przypisywane do parametrów blokowych.

5.3.1. Iteratory numeryczne

Rdzenne API Ruby udostępnia kilka standardowych iteratorów. Metoda `loop` z modułu `Kernel` działa jak nieskończona pętla wielokrotnie wykonująca związaną z nią blok kodu, dopóki nie wykona instrukcji `return`, `break` lub jakiejś innej, która spowoduje wyjście z pętli.

Klasa `Integer` definiuje trzy często używane iteratory. Metoda `upto` wywołuje zвязany z nią blok kodu po jednym razie dla każdej liczby całkowitej z przedziału pomiędzy liczbą, na rzecz której została wywołana, a liczbą przekazaną do niej jako argument. Na przykład:

```
4.upto(6) { |x| print x }    # => Drukuję "456".
```

Iteratory, które nie iterują

W niniejszej książce mianem **iteratorka** określone są wszystkie metody używające instrukcji `yield`. Nie muszą one koniecznie służyć jako funkcje iteracyjne bądź pętlowe⁴. Przykładem tego może być metoda `tap` zdefiniowana (w Ruby 1.9 oraz 1.8.7) w klasie `Object`. Wywołuje ona związaną z nią blok kodu jeden raz, przekazując odbiorcę jako jedyny argument. Następnie zwraca odbiorcę. Jest przydatna do „wpychania się” w łańcuchach wywołań metod jak w poniższym kodzie, który używa metody `tap` do wysyłania na wyjście komunikatów debugowania:

```
chars = "hello world".tap { |x| puts "original object: #{x.inspect}"}
  .each_char          .tap { |x| puts "each_char returns: #{x.inspect}"}
  .to_a              .tap { |x| puts "to_a returns: #{x.inspect}"}
  .map { |c| c.succ } .tap { |x| puts "map returns: #{x.inspect}" }
  .sort               .tap { |x| puts "sort returns: #{x.inspect}" }
```

Innym częstym zastosowaniem iteratorów jest automatyczna dealokacja zasobów. Na przykład jako iterator może być użyta metoda `File.open`. Otwiera ona plik o podanej nazwie i tworzy jego reprezentację w obiekcie klasy `File`. Jeśli z wywołaniem tym nie jest związany żaden blok kodu, zostaje zwrocony tylko obiekt klasy `File`, a zadanie zamknięcia otwartego pliku pozostaje w sferze obowiązków kodu, który ją wywołał. Jeżeli natomiast z metodą ta jest związany jakiś blok kodu, obiekt klasy `File` zostaje do niego przekazany i plik zostaje automatycznie zamknięty po zwrocienniu przez ten blok wartości. Dzięki temu pliki są zawsze zamykane, a programista nie musi pamiętać o mniej ważnych szczegółach. W przypadku gdy z wywołaniem metody `File.open` jest związany jakiś blok kodu, wartość zwrotna metody nie należy do klasy `File`, ale jest dowolną wartością zwroconą przez blok.

Jak widać, instrukcja `yield` przekazuje każdą liczbę całkowitą do bloku kodu. W iteracji brane są pod uwagę punkty początkowy i końcowy. Ogólnie rzecz biorąc, instrukcja `n.upto(m)` wywołuje swój blok kodu $m-n+1$ razy.

Metoda `downto` działa podobnie do `upto` z tym wyjątkiem, że iteruje od większej liczby do mniejszej.

Metoda `Integer.times` wywołana na rzecz obiektu klasy `Integer` n wywołuje swój blok kodu n razy, przekazując w kolejnych iteracjach wartości od 0 do $n-1$. Na przykład:

```
3.times { |x| print x }    #=> Drukuje "012".
```

Ogólnie wywołanie `n.times` jest równoważne z `0.upto(n-1)`.

Do iteracji przy użyciu liczb zmiennoprzecinkowych służy bardziej skomplikowana metoda o nazwie `step` zdefiniowana w klasie `Numeric`. Na przykład poniższy iterator zaczyna iterację od 0 i w kolejnych powtórzeniach zwiększa wartość o 0.1, aż dojdzie do wartości `Math::PI`:

```
0.step(Math::PI, 0.1) { |x| puts Math.sin(x) }
```

⁴ W japońskiej społeczności skupionej wokół języka Ruby określenie „iteratorka” wyszło całkiem z użytku, ponieważ oznacza ono iterację, która nie zawsze jest wymagana. Wyrażenie typu „metoda, która przyjmuje związkę z nią blok kodu” jest bardziej rozwlekłe, ale też bardziej precyzyjne.

5.3.2. Obiekty przeliczalne

Klasy Array, Hash i Range, a także wiele innych definiują iterator each, który przekazuje każdy element kolekcji do związanego z wywołaniem bloku kodu. Jest to prawdopodobnie najczęściej używany iterator w języku Ruby. Jak już było wspomniane wcześniej, pętla for działa tylko na obiektach, które udostępniają metodę each. Przykładowe iteratory each:

```
[1,2,3].each { |x| print x }      #=> Drukuję "123".  
(1..3).each { |x| print x }      #=> Drukuję "123"— to samo, co 1.upto(3).
```

Iterator each nie jest jednak przeznaczony tylko dla tradycyjnych klas struktur danych. Klasa IO definiuje iterator each, który zwraca wiersze tekstu wczytane z obiektu wejścia lub wyjścia. W związku z tym plik można przetworzyć za pomocą poniższego kodu:

```
File.open(filename) do |f|          # Otwarcie podanego pliku, przekazanie jako f.  
  f.each { |line| print line }      # Wydruk każdego wiersza z f.  
end                                # Koniec bloku i zamknięcie pliku.
```

Większość klas definiujących metodę each zawiera także moduł Enumerable, który definiuje kilka bardziej wyspecjalizowanych iteratorów zaimplementowanych na podstawie tej metody. Jednym z nich jest each_with_index, dzięki któremu można w poprzednim przykładzie włączyć numerowanie wierszy:

```
File.open(filename) do |f|  
  f.each_with_index do |line,number|  
    print "#{number}: #{line}"  
  end  
end
```

Do najpowszechniej używanych iteratorów modułu Enumerable należą metody collect, select, reject i inject. Metoda collect (znana też jako map) wykonuje swój blok kodu dla każdego elementu obiektu przeliczalnego i zbiera zwrócone wartości w tablicy:

```
squares = [1,2,3].collect { |x| x*x}  #=> [1,4,9].
```

Metoda select wywołuje związaną z nią blok kodu dla każdego elementu w przeliczalnym obiekcie i zwraca tablicę elementów, dla których blok zwrócił wartość inną niż false albo nil. Na przykład:

```
evens = (1..10).select { |x| x%2 == 0} #=> [2,4,6,8,10].
```

Metoda reject jest przeciwnieństwem metody select — zwraca tablicę elementów, dla których blok zwrócił wartość nil lub false. Na przykład:

```
odds = (1..10).reject { |x| x%2 == 0} #=> [1,3,5,7,9].
```

Metoda inject jest nieco bardziej skomplikowana od pozostałych. Wywołuje związaną z nią blok kodu z dwoma argumentami. Pierwszy argument to wartość zbiorcza wcześniejszych iteracji. Drugi argument jest kolejnym elementem obiektu przeliczalnego. Wartością zwrotną tego bloku jest pierwszy argument blokowy następnej iteracji lub wartość zwrotnej iteratora zwrócona po ostatniej iteracji. Początkową wartością zmiennej akumulacyjnej jest argument metody inject, jeśli istnieje, lub pierwszy element obiektu przeliczalnego (w tym przypadku blok jest wywoływany tylko jeden raz dla dwóch pierwszych elementów). Sposób działania metody inject łatwiej jest zrozumieć na przykładach:

```
data = [2, 5, 3, 4]  
sum = data.inject { |sum, x| sum + x }      #=> 14  (2+5+3+4).  
floatprod = data.inject(1.0) { |p, x| p*x }  #=> 120.0 (1.0*2*5*3*4).  
max = data.inject { |m, x| m>x ? m : x }    #=> 5  (największy element).
```

Więcej informacji na temat modułu Enumerable i jego iteratorów znajduje się w podrozdziale 9.5.1.

5.3.3. Pisanie niestandardowych iteratorów

Cechą charakterystyczną metody iteracyjnej jest to, że wywołuje ona blok kodu związany z jej wywołaniem. Jest to wykonywane za pomocą instrukcji `yield`. Poniższa metoda jest bardzo prostym iteratorem, który dwukrotnie wywołuje swój blok kodu:

```
def twice
  yield
  yield
end
```

Aby przekazać do bloku wartości argumentów, należy po instrukcji `yield` wstawić listę wyrażeń oddzielonych przecinkami. Podobnie jak w wywołaniu metody, wartości argumentów mogą opcjonalnie być umieszczone w nawiasach. Poniższy prosty iterator demonstruje użycie instrukcji `yield`:

```
# Niniejsza metoda pobiera blok. Generuje n wartości w formie
# m*i + c dla i należącego do zbioru 0..n-1 oraz przekazuje je pojedynczo do
# związanego z nią bloku.
def sequence(n, m, c)
  i = 0
  while(i < n)      # Powtóżenie n razy.
    yield m*i + c   # Wywołanie bloku i przekazanie do niego wartości.
    i += 1           # Zwiększenie i za każdym razem.
  end
# Wywołanie zdefiniowanej metody z blokiem.
# Drukuje wartości 1, 6 i 11.
sequence(3, 5, 1) { |y| puts y }
```

Nomenklatura: `yield` i iteratory

W zależności od posiadanego doświadczenia programistycznego niektórym Czytelnikom nazwy `yield` i iterator mogą wydawać się mylące. Prezentowana wcześniej metoda `sequence` jest doskonałym przykładem, dlaczego metoda `yield` ma taką nazwę, a nie inną. Po obliczeniu każdej liczby w szeregu metoda ta przekazuje sterowanie (ang. `yield`) i obliczoną liczbę do bloku, aby zawarty w nim kod mógł ją przetworzyć. Nie zawsze jest to jednak takie jasne. Czasami może wydawać się, że to blok przekazuje wynik z powrotem do metody, która go wywołała.

Metoda taka jak `sequence`, wymagająca bloku i wywołująca go wielokrotnie, nazywa się iteratorem, ponieważ wygląda i działa jak pętla. Jest to coś nowego dla programistów na przykład Javy, w której iteratory są obiektami. W Javie kontrolę ma kod kliencki używający iteratora i pobierający z niego wartości, kiedy ich potrzebuje. W Ruby kontrolę ma iterator przekazujący wartości do wymagającego ich bloku.

To zagadnienie nomenklaturowe jest związane z rozróżnieniem iteratorów wewnętrznych i zewnętrznych; zostało ono opisane dalej w tym podrozdziale.

Oto jeszcze jeden przykład iteratora, który przekazuje dwa argumenty do swojego bloku. Warto zauważyć, że w jego implementacji użyto innego iteratora na potrzeby wewnętrzne:

```
# Generowanie n punktów równomiernie rozmieszczonych na obwodzie
# koła o promieniu r i środku w punkcie (0,0). Współrzędne x i y każdego punktu
# są przekazywane do odpowiedniego bloku kodu.
def circle(r,n)
  n.times do |i|      # Niniejsza metoda została zaimplementowana przy użyciu bloku.
    angle = Math::PI * 2 * i / n
```

```

        yield r*Math.cos(angle), r*Math.sin(angle)
    end
end
# Niniejsze wywołanie powyższego iteratora drukuje:
# (1.00, 0.00) (0.00, 1.00) (-1.00, 0.00) (-0.00, -1.00).
circle(1,4) { |x,y| printf "(%.2f, %.2f) ", x, y }

```

Użycie słowa kluczowego `yield` w znacznym stopniu przypomina wywołanie metody (szczegółowe informacje na temat wywoływanego metod znajdują się w rozdziale 6.). Nawiązy otaczające argumenty są opcjonalne. Za pomocą `*` można rozbić tablicę na poszczególne argumenty. Instrukcja `yield` pozwala nawet na przekazanie literała tablicy asocjacyjnej bez otaczających ją klamer. Jednak w przeciwieństwie do wywołania metody po wyrażeniu `yield` nie może znajdować się blok kodu. Nie można przekazać bloku do bloku.

Jeśli metoda jest wywoływana bez bloku, błędem jest używanie przez nią instrukcji `yield`, ponieważ nie ma gdzie przekazywać sterowania. Czasami potrzebna jest metoda, która przekazuje sterowanie i wartości do bloku kodu, jeżeli jakiś istnieje, ale wykonuje jakieś domyślne działania (inne niż spowodowanie błędu), gdy zostanie wywołana bez żadnego bloku. W tym celu należy sprawdzić obecność bloku związanego z wywołaniem metody za pomocą metody `block_given?`, która, tak jak jej synonim `iterator?`, należy do modułu `Kernel`, a więc działa jak funkcja globalna. Oto stosowny przykład:

```

# Zwraca tablicę n elementów w formie m*i+c.
# Jeśli blok został podany, każdy z tych elementów jest do niego przekazywany.
def sequence(n, m, c)
  i, s = 0, []
  while(i < n)
    # Inicjacja zmiennych.
    y = m*i + c
    # Powtarzanie n razy.
    yield y if block_given?
    # Przekazanie sterowania i wartości, jeżeli jest blok.
    s << y
    # Zapisanie wartości.
    i += 1
  end
  s
end
# Zwrot tablicy wartości.

```

5.3.4. Enumeratory

Enumerator to przeliczalny obiekt (`Enumerable`), którego zadaniem jest iteracja po innym obiekcie. Aby używać enumeratorów w Ruby 1.8 konieczne jest dodanie do pliku źródłowego kodu `require 'enumerator'`. W Ruby 1.9 (a także w 1.8.7) obiekty te są wbudowane, a więc nie ma potrzeby stosowania metody `require` (później dowiesz się, że wbudowane enumeratory Ruby są znacznie bardziej funkcjonalne niż te z biblioteki `enumerator`).

Enumerator należą do klasy `Enumerable::Enumerator`. Mimo że można je tworzyć bezpośrednio za pomocą słowa kluczowego `new`, zazwyczaj do tego celu wykorzystywana jest metoda `to_enum` lub jej synonim `enum_for` — obie należą do klasy `Object`. Jeżeli nie zostaną podane żadne argumenty, metoda `to_enum` zwraca enumerator, którego metoda `each` wywołuje metodę `each` obiektu docelowego. Założmy, że masz tablicę i metodę, która przyjmuje przeliczalny obiekt. Nie chcesz przekazywać samego obiektu tablicy, ponieważ może on zostać zmodyfikowany, a Ty nie masz pewności, czy metoda ta go nie zmodyfikuje. Zamiast wykonywać głęboką kopię tablicy, lepiej jest wywołać na jej rzecz metodę `to_enum` i zamiast samej tablicy przekazać uzyskany w ten sposób enumerator. W ten sposób tworzony jest przeliczalny, ale niemodyfikowalny obiekt zastępczy dla tablicy:

```
# Wywołanie metody z enumeratorem zamiast tablicy, którą można byłoby modyfikować.  
# Jest to przydatna strategia obronna pozwalająca uniknąć błędów.  
process(data.to_enum) # Zamiast process(data).
```

Do metody `to_enum` można także przekazywać argumenty, chociaż w takim przypadku bardziej naturalne wydaje się być użycie jej synonimu `enum_for`. Pierwszym argumentem powinien być symbol identyfikujący metodę iteracyjną. Metoda `each` powstałego obiektu klasy `Enumerator` wywoła podaną metodę oryginalnego obiektu. Wszystkie pozostałe argumenty metody `enum_for` zostaną przekazane do tamtej metody. W Ruby 1.9 (oraz 1.8.7) klasa `String` nie jest typu `Enumerable`, ale udostępnia trzy metody iteracyjne: `each_char`, `each_byte` oraz `each_line`. Założymy, że chcesz użyć metody typu `Enumerable`, na przykład `map`, która powinna być oparta na iteratorze `each_char`. W tym celu należy utworzyć enumerator:

```
s = "hello"  
s.enum_for(:each_char).map {|c| c.succ} # => ["i", "f", "m", "m", "p"].
```

W Ruby 1.9 w większości przypadków nie trzeba jawnie wywoływać metody `to_enum` ani `enum_for`, jak to było robione w dotychczasowych przykładach. Powodem tego jest to, że wbudowane metody iteracyjne w Ruby 1.9 (do których należą iteratory numeryczne `times`, `upto`, `downto` i `step` oraz `each` i pokrewne metody modułu `Enumerable`) automatycznie zwracają enumerator, jeśli są wywoływane bez żadnego bloku. W związku z tym, aby do metody przekazać enumerator tablicy, a nie samą tablicę, można wywołać tylko metodę `each`:

```
process(data.each_char) # Zamiast process(data).
```

Składnia ta jest jeszcze bardziej naturalna przy użyciu aliasu `chars` w miejscu metody `each_char`. Żeby na przykład znaki łańcucha przekonwertować na tablicę znaków, należy użyć wywołania `.chars.map`:

```
"hello".chars.map {|c| c.succ} # => ["i", "f", "m", "m", "p"].
```

Oto jeszcze kilka przykładów z użyciem obiektów enumeratora zwróconych przez metody iteracyjne. Należy zauważać, że nie tylko metody iteracyjne zdefiniowane w module `Enumerable` mogą zwracać obiekty enumeracyjne. To samo robią iteratory numeryczne jak `times` i `upto`:

```
enumerator = 3.times # Obiekt enumeracyjny.  
enumerator.each {|x| print x} # Drukuje "012".  
# Metoda downto zwraca enumerator z metodą select.  
10.downto(1).select {|x| x%2==0} # => [10, 8, 6, 4, 2]  
# Metoda iteracyjna each_byte zwraca enumerator z metodą to_a.  
"hello".each_byte.to_a # => [104, 101, 108, 108, 111]
```

Można to działanie zduplikować we własnych metodach, zwracając `self.to_enum`, kiedy nie podano żadnego bloku. Na przykład poniżej znajduje się wersja prezentowanej wcześniej metody iteracyjnej `twice` zwracającej enumerator, jeśli nie dostarczono żadnego bloku:

```
def twice  
  if block_given?  
    yield  
    yield  
  else  
    self.to_enum(:twice)  
  end  
end
```

W Ruby 1.9 obiekty enumeracyjne udostępniają metodę `with_index` niedostępna w module enumeracyjnym Ruby 1.8. Metoda `with_index` zwraca nowy enumerator, który dodaje do iteracji parametr indeksu. Na przykład poniższa procedura zwraca enumerator przekazujący znaki łańcucha i ich indeksy w tym łańcuchu:

```
enumerator = s.each_char.with_index
```

Na koniec należy pamiętać, że enumeratory zarówno w Ruby 1.8, jak i 1.9 są obiektami przeliczalnymi, których można używać w połączeniu z pętlą `for`. Na przykład:

```
for line, number in text.each_line.with_index
  print "#{number+1}: #{line}"
end
```

5.3.5. Iteratory zewnętrzne

Dotychczasowy opis enumeratorów koncentrował się na użyciu ich jako przeliczalnych obiektów zastępczych. Jednak w Ruby 1.9 (a także w 1.8.7, choć w tej wersji implementacja nie jest tak efektywna) mają one jeszcze jedno bardzo ważne zastosowanie — są **iteratorem zewnętrznym**. Za pomocą enumeratora można przejść iteracyjnie przez elementy kolekcji, wywołując wielokrotnie metodę `next`. Kiedy wyczerpią się elementy, zgłasza ona wyjątek `StopIteration`:

```
iterator = 9.downto(1)                      # Enumerator w roli zewnętrznego iteratora.
begin                                         # Aby można było użyć poniżej metody rescue.
  print iterator.next while true             # Wielokrotne wywołanie metody next.
rescue StopIteration                         # Kiedy nie ma więcej wartości,
  puts "...start!"                          # następuje spodziewana, niewyjątkowa sytuacja.
end
```

Iteratory wewnętrzne a iteratory zewnętrzne

Bardzo jasna definicja i precyzyjne porównanie iteratorów wewnętrznych i zewnętrznych znajdują się w książce na temat wzorców projektowych napisanej przez bandę czterech autorów (*Gang of Four*)⁵:

Najważniejsze jest ustalenie, kto kontroluje iterację — iterator czy używający go klient. Jeśli ma miejsce druga z wymienionych sytuacji, masz do czynienia z **iteratorem zewnętrznym**. Gdy natomiast iterację kontroluje iterator, masz do czynienia z **iteratorem wewnętrzny**m. Klienci używający zewnętrznego iteratora muszą popychać proces iteracji do przodu i jawnie żądać od iteratora kolejnych elementów. Dla kontrastu — klient przekazuje wewnętrzemu iteratorowi zadanie do wykonania, a ten stosuje tę operację do każdego z elementów...

Iteratory zewnętrzne są bardziej elastyczne niż iteratory wewnętrzne. Na przykład sprawdzenie, czy dwie kolekcje są równe, za pomocą iteratora zewnętrznego jest łatwe, ale praktycznie niemożliwe przy użyciu iteratora wewnętrznego... Z drugiej strony iteratory wewnętrzne są łatwiejsze w użyciu, ponieważ definiują za użytkownika procedury iteracyjne.

W języku Ruby do iteratorów wewnętrznych należy na przykład metoda `each`. Kontroluje ona iterację i „popycha” wartości do bloku kodu związanego z wywołaniem metody. Enumeratory posiadają metodę `each` pełniąą rolę wewnętrznego iteratora, ale od wersji 1.9 metody te działają także jako iteratory zewnętrzne — kod kliencki może sekwencyjnie „wyciągać” wartości z enumeratora za pomocą metody `next`.

Iteratory zewnętrzne są bardzo łatwe w użyciu — za każdym razem, kiedy potrzebny jest nowy element, należy wywołać metodę `next`. Jeśli wyczerpią się wszystkie elementy, metoda `next` zgłosi wyjątek `StopIteration`. Może się to wydawać nieco zaskakujące — wyjątek jest

⁵ Design Patterns: Elements of Reusable Object-Oriented Software, Gamma, Helm, Johnson i Vlissides (Addison-Wesley).

zgłaszanego w spodziewanej sytuacji końca zbioru, a nie w wyniku niespodziewanego czy wyjątkowego zdarzenia (wyjątek `StopIteration` jest potomkiem klasy `StandardError` i `IndexError`; należy zauważyć, że jest to jedna z nielicznych klas wyjątków, które nie mają w nazwie słowa `Error`). Ta technika zewnętrznej iteracji została przez język Ruby odziedziczona po Pythonie. Dzięki potraktowaniu końca pętli jako wyjątku logika pętlowa staje się niezwykle prosta. Nie trzeba szukać w wartości zwrotnej metody `next` żadnej specjalnej wartości oznaczającej koniec iteracji oraz nie ma potrzeby wywoływanego żadnego predykatu typu `next?` przed każdym wywołaniem metody `next`.

Aby uprościć wykonywanie operacji pętlowych przy użyciu zewnętrznych iteratorów, w metodzie `Kernel.loop` dodano (w Ruby 1.9) jawną klauzulę `rescue` i sprawiono, że w chwili powstania wyjątku `StopIteration` kończy ona działanie. Dzięki temu zaprezentowaną wcześniej procedurę odliczającą można zapisać w prostszy sposób:

```
iterator = 9.downto(1)
loop do
  print iterator.next      # Powtarzanie aż do wystąpienia wyjątku StopIteration.
end
puts "...start!"          # Wydruk kolejnego elementu.
```

Wiele zewnętrznych iteratorów można cofnąć do punktu wyjściowego za pomocą metody `rewind`. Należy jednak pamiętać, że nie działa ona na wszystkich enumeratorkach. Jeśli enumerator działa, opierając się na obiekcie na przykład klasy `File` wczytującym plik wiersz po wierszu, wywołanie metody `rewind` nie spowoduje uruchomienia iteracji od początku. Ogólnie rzecz biorąc, jeżeli nowe wywołania metody `each` na rzecz leżącego u podłożu obiektu `Enumerable` nie powodują ponownego uruchomienia iteracji, wywołanie metody `rewind` również tego nie zrobi.

Po rozpoczęciu wewnętrznej iteracji (to jest po pierwszym wywołaniu metody `next`) enumeratorka nie można sklonować ani zduplikować. Zazwyczaj sklonowanie enumeratorka możliwe jest przed wywołaniem metody `next`, po zgłoszeniu wyjątku `StopIteration` albo po wywołaniu metody `rewind`.

W typowej sytuacji enumeratory z metodą `next` są tworzone z obiektów `Enumerable`, które udostępniają metodę `each`. Jeśli z jakiegoś powodu zostanie zdefiniowana klasa udostępniająca metodę `next` do iteracji zewnętrznej zamiast metody `each` do iteracji wewnętrznej, z łatwością można zaimplementować metodę `each` na podstawie metody `next`. Zamiana klasy z zewnętrznym iteratorem implementującą metodę `next` na klasę `Enumerable` jest tak łatwa jak dodanie modułu domieszkowego (za pomocą metody `include` — zobacz podrozdział 7.5), jak poniżej:

```
module Iterable
  include Enumerable
  def each
    loop { yield self.next }
  end
end
```

Innym sposobem użycia zewnętrznego iteratorka jest przekazanie go do wewnętrznej metody iteracyjnej:

```
def iterate(iterator)
  loop { yield iterator.next }
end
iterate(9.downto(1)) {|x| print x }
```

Cytowany wcześniej fragment książki o wzorcach projektowych zawierał aluzję do jednej z klu-czowych własności iteratorów zewnętrznych — rozwiązującą one problem równoległego przypi-sywania. Założymy, że masz dwie przeliczalne kolekcje, których elementy muszą zostać zbadane iteracyjnie w parach — najpierw pierwsze elementy każdej z kolekcji, potem drugie elementy itd. Przy braku zewnętrznego iteratora konieczne jest przekonwertowanie jednej z tych ko-lekcji na tablicę (za pomocą metody `to_a` zdefiniowanej w module `Enumerable`), aby móc uzyskać dostęp do jej elementów podczas iteracji drugiej kolekcji za pomocą metody `each`.

Listing 5.1 przedstawia implementację trzech metod iteracyjnych. Każda z nich przyjmuje dowolną liczbę przeliczalnych obiektów i iteruje je w inny sposób. Jedna z nich to prosty iterator sekwencyjny używający tylko wewnętrznych iteratorów, pozostałe dwie to iteracje równoległe, które mogą być wykonywane wyłącznie przy użyciu własności zewnętrznych iteratorów Ruby 1.9.

Listing 5.1. Iteracja równoległa przy użyciu iteratorów zewnętrznych

```
# Wywołanie metody each po kolejno w każdej z kolekcji.
# Nie jest to iteracja równoległa, a więc nie jest wymagany enumerator.
def sequence(*enumerables, &block)
  enumerables.each do |enumerable|
    enumerable.each(&block)
  end
end
# Iteracja przez podane kolekcje z przeplataniem ich elementów.
# Nie można tego zrobić szybko bez użycia iteratorów zewnętrznych.
# Warto zauważyć rzadki przypadek użycia klauzuli else w bloku begin-rescue.
def interleave(*enumerables)
  # Konwersja przeliczalnych kolekcji na tablicę enumeratorów.
  enumerators = enumerables.map { |e| e.to_enum }
  # Pętla działa tak długo, aż wyczerpią się enumeratory.
  until enumerators.empty?
    begin
      e = enumerators.shift          # Pobranie pierwszego enumeratora.
      yield e.next                    # Pobranie kolejnego elementu i przekazanie go do bloku.
    rescue StopIteration            # Jeśli nie ma więcej elementów, nic się nie dzieje.
    else                            # Jeżeli nie wystąpił żaden wyjątek,
      enumerators << e           # enumerator zostaje zwrotnie powrotem.
    end
  end
end
# Iteracja przez określone kolekcje ze zwrotem krotek wartości,
# po jednej wartości z każdej kolekcji. Zobacz też Enumerable.zip.
def bundle(*enumerables)
  enumerators = enumerables.map { |e| e.to_enum }
  loop { yield enumerators.map { |e| e.next } }
end
# Przykłady działania powyższych metod iteracyjnych.
a,b,c = [1,2,3], 4..6, 'a'...'e'
sequence(a,b,c) { |x| print x }    # Drukuje "123456abcde".
interleave(a,b,c) { |x| print x }  # Drukuje "14a25b36cde".
bundle(a,b,c) { |x| print x }     # '[1, 4, "a"]/2, 5, "b"/3, 6, "c"]'.
```

Metoda `bundle` użyta w powyższym listingu jest podobna do metody `Enumerable.zip`. W Ru-by 1.8 metoda `zip` musi wpierw przekonwertować swoje przeliczalne argumenty na tablice, a na-stępnie użyć ich podczas iteracji przez przeliczalny obiekt, na rzecz którego została wywo-la-na. Natomiast w Ruby 1.9 metoda `zip` może używać iteratorów zewnętrznych. To z reguły przyspiesza działanie operacji i oszczędza miejsce w pamięci, a także umożliwia używanie nieograniczonych kolekcji, które nie mogą zostać przekonwertowane na tablice o określ-o-nych rozmiarach.

5.3.6. Iteracja i modyfikowanie współbieżne

Ogólnie rdzenne klasy kolekcyjne języka Ruby iterują przez żywe obiekty, a nie przez ich kopie, i nie sprawdzają ani też w żaden sposób nie zabezpieczają się przed modyfikacją kolekcji w trakcie iteracji po niej. Jeśli na przykład zostanie wywołana metoda `each` tablicy, a w bloku związanym z tym wywołaniem znajduje się wywołanie metody `shift` tej samej tablicy, wyniki iteracji mogą być zaskakujące:

```
a = [1,2,3,4,5]
a.each { |x| puts "#{x},#{a.shift}" } # Drukuje "1,1\n3,2\n5,3".
```

Podobnie zaskakujące działanie można zaobserwować, kiedy jeden wątek modyfikuje kolekcję, podczas gdy drugi iteruje po niej. Jednym ze sposobów na uniknięcie takiej sytuacji jest utworzenie kopii zapasowej kolekcji przed iteracją po niej. Na przykład poniższy kod dodaje metodę `each_in_snapshot` do modułu `Enumerable`:

```
module Enumerable
  def each_in_snapshot &block
    snapshot = self.dup      # Utworzenie prywatnej kopii obiektu Enumerable.
    snapshot.each &block     # Iteracja po tej kopii.
  end
end
```

5.4. Bloki

Bloki mają fundamentalne znaczenie przy używaniu iteratorów. W poprzednim podrozdziale skoncentrowaliśmy się na iteratorach jako pewnego rodzaju konstrukcjach pętlowych. Bloki cały czas przewijały się przez podrozdział, ale nie były jego tematem. Teraz zwracamy naszą uwagę właśnie na bloki. Kolejne podrozdziały zawierają opis:

- sposobów wiązania bloku z wywołaniem metody,
- wartości zwrotnej bloku,
- zasięgu zmiennych w blokach,
- różnicy między parametrami blokowymi a parametrami metod.

5.4.1. Składnia bloku

Bloki nie mogą występować samodzielnie, muszą być związane z wywołaniem jakiejś metody. Jednak blok można umieścić po każdej metodzie. Jeśli metoda ta nie jest iteratorem i nigdy nie wywołuje bloku za pomocą instrukcji `yield`, blok zostaje po cichu zignorowany. Bloki są ograniczane nawiasami klamrowymi lub słowami kluczowymi `do` i `end`. Otwierający nawias klamrowy lub słowo kluczowe `do` musi znajdować się w tym samym wierszu co wywołanie metody. W przeciwnym razie interpreter potraktuje koniec wiersza jako koniec instrukcji i wywoła metodę bez bloku:

```
# Drukuję liczby od 1 do 10.
1.upto(10) { |x| puts x }      # Wywołanie i blok w jednym wierszu z klamrami.
1.upto(10) do |x|              # Blok ograniczony słowami kluczowymi do i end.
  puts x
end
1.upto(10)                      # Żaden blok nie został podany.
{ |x| puts x }                  # Błąd składni: blok nie znajduje się za wywołaniem.
```

Powszechną konwencją jest używanie nawiasów klamrowych, gdy blok mieści się w jednym wierszu, a słów kluczowych do i end, kiedy zajmuje on kilka wierszy. Nie jest to jednak tylko sprawą konwencji. Interpreter Ruby silnie wiąże nawias { ze znajdującym się przed nim tokenem. Jeśli nawias oddzielający argumenty metody zostanie pominięty, a blok zostanie oddzielony od wywołania klamrami, blok ten zostanie związany z ostatnim argumentem metody, a nie z nią samą, co najczęściej jest sytuacją niepożądaną. Aby tego uniknąć, argumenty należy otaczać nawiasami lub bloki umieszczać między słowami kluczowymi do i end:

```
1.upto(3) { |x| puts x }      # Nawiasy i klamry działają dobrze.  
1.upto 3 do |x| puts x end # Brak nawiasów, blok otoczony słowami kluczowymi do i end.  
1.upto 3 { |x| puts x }      # Błąd składni: próba przekazania bloku do liczby 3!
```

Bloki, podobnie jak metody, mogą mieć parametry. Parametry blokowe są rozdzielane przecinkami i otaczane parą pionowych kresek (|), ale poza tym są bardzo podobne do parametrów metod.

```
# Iterator Hash.each przekazuje dwa argumenty do swojego bloku.  
hash.each do |key, value|    # Dla każdej pary (key,value) w tablicy asocjacyjnej.  
  puts "#{key}: #{value}"    # Drukuj klucz i jego wartość.  
end                          # Koniec bloku.
```

Powszechnie przyjęta konwencja nakazuje parametry blokowe wpisywać w tym samym wierszu co wywołanie metody i otwierająca klamra lub słowo kluczowe do, ale nie jest to wymóg składni.

5.4.2. Wartość bloku

W prezentowanych dotychczas przykładach iteratorów metoda iteracyjna przekazywała wartości do swojego bloku, ale ignorowała te, które zostały przez niego zwrócone. Nie zawsze tak jednak jest. Weźmy na przykład metodę Array.sort. Jeśli z jej wywołaniem zostanie związany jakiś blok, zwróci ona do niego pary elementów i to do bloku będzie należało ich posortowanie. Wartość zwrotna bloku (-1, 0 lub 1) oznacza kolejność dwóch argumentów. Wartość ta jest dostępna w metodzie iteracyjnej jako wartość zwrotna instrukcji yield.

„Wartość zwrotna” bloku to wartość ostatniego w nim wyrażenia. Zatem do posortowania tablicy słów od najdłuższego do najkrótszego można napisać następującą procedurę:

```
# Blok pobiera dwa słowa i „zwraca” je w porządku wg długości.  
words.sort! { |x,y| y.length <=> x.length }
```

Wyrażenie „wartość zwrotna” umieszczone są w cudzysłowach z bardzo ważnego powodu — do zwracania wartości z bloku nie należy normalnie używać słowa kluczowego return. Słowo to wewnętrz bloku powoduje zwrócenie wartości przez metodę go zawierającą (nie metodę iteracyjną, a tę, w której skład wchodzi blok). Oczywiście czasami właśnie to powinno się stać. Nie należy jednak używać słowa kluczowego return, aby przekazać wartość zwrotną bloku do metody, która wywołała instrukcję yield. Aby zmusić blok do zwrócenia wartości do wywołującej go metody przed dotarciem do ostatniego wyrażenia lub do zwrócenia więcej niż jednej wartości, należy zamiast return użyć słowa kluczowego next (informacje na temat słów kluczowych return, next i związanego z nimi break znajdują się w podrozdziale 5.5). Poniżej przedstawiony jest przykład zwrócenia wartości z bloku przy użyciu instrukcji next:

```
array.collect do |x|  
  next 0 if x == nil  # Przedwczesny zwrot wartości, jeśli x ma wartość nil.  
  next x, x*x        # Zwrot dwóch wartości.  
end
```

Warto pamiętać, że taki sposób użycia słowa kluczowego `next` nie jest zbyt powszechny i powyższy kod można z łatwością przepisać bez niego:

```
array.collect do |x|
  if x == nil
    0
  else
    [x, x*x]
  end
end
```

5.4.3. Bloki a zasięg zmiennych

Bloki definiują nowy zakres widoczności dla zmiennych — zmienne utworzone w bloku istnieją tylko w nim i poza nim nie są zdefiniowane. Należy jednak pamiętać, że zmienne lokalne w metodach są dostępne we wszystkich blokach tych metod. Jeśli zatem w bloku znajduje się przypisanie wartości do zmiennej, która jest już poza nim zdefiniowana, nie powoduje to utworzenia nowej zmiennej o zasięgu blokowym, a przypisanie nowej wartości do już istniejącej zmiennej. Czasami jest to dokładnie to, czego oczekuje programista:

```
total = 0
data.each { |x| total += x }      # Sumowanie elementów tablicy data.
puts total                      # Wydruk tej sumy.
```

Niekiedy jednak zmiana wartości zmiennych z szerszego zakresu nie jest zamierzona, tylko odbywa się nieumyślnie. Problem ten jest w szczególności związany z parametrami blokowymi w Ruby 1.8. W tej wersji języka, jeśli parametr blokowy ma taką samą nazwę jak istniejąca zmienna, wywołanie bloku powoduje przypisanie wartości do tamtej zmiennej, a nie utworzenie nowej o zasięgu blokowym. Na przykład poniższy kod sprawia problemy, ponieważ użyto w nim identyfikatora `i` o takiej samej nazwie jak parametry dwóch zagnieżdzonych bloków:

```
1.upto(10) do |i|                  # 10 wierszy,
  1.upto(10) do |i|                  # każdy zawiera 10 kolumn.
  print "#{i} "                     # Drukowanie numeru kolumny.
end
print " ==> Wiersz #{i}\n"        # Próba wydrukowania numeru wiersza, ale pobranie numeru kolumny.
end
```

W Ruby 1.9 jest inaczej — parametry blokowe mają zawsze zasięg lokalny ograniczony do swojego bloku i wywołania bloku nie powodują przypisania wartości do istniejących zmiennych. Jeśli interpreter zostanie uruchomiony z opcją `-w`, będzie ostrzegał o parametralach blokowych o takiej samej nazwie jak istniejące zmienne. To pomaga uniknąć napisania kodu, który działa inaczej w Ruby 1.8, a inaczej w Ruby 1.9.

Ruby 1.9 ma jeszcze jedną ważną cechę. Dodano w nim rozszerzenie składni bloków pozwalające na deklarację zmiennych blokowych, które będą zawsze lokalne, nawet jeśli zmienią o takiej samej nazwie istnieje w zakresie nadziedzonym. W tym celu po liście parametrów blokowych należy umieścić średnik i listę zmiennych lokalnych oddzielonych przecinkami. Na przykład:

```
x = y = 0                      # Zmienne lokalne.
1.upto(4) do |x; y|              # Zmienne x i y mają zasięg blokowy.
                                  # Zmienne x i y „zasłaniają” zewnętrzne zmienne.
  y = x + 1                      # Zmieniona y użyta jako zmieniona początkowa.
  puts y*x                      # Drukuje 4, 9, 16, 25.
end
[x, y]                          # => [0,0]: blok nie zmienia tych wartości.
```

W powyższym kodzie `x` jest parametrem blokowym — otrzymuje wartość w chwili wywołania bloku za pomocą instrukcji `yield`. Zmienna `y` jest zmienną lokalną blokową. Nie otrzymuje żadnej wartości od instrukcji `yield`, ale dopóki blok nie przypisze jej jakiejś wartości, ma wartość `nil`. Celem deklarowania tych zmiennych lokalnych blokowych jest zagwarantowanie, że nie zostanie przypadkowo zmieniona wartość jakiejś istniejącej zmiennej (może się to zdarzyć na przykład po wycięciu i wklejeniu kodu z jednej metody do innej). Jeśli interpreter Ruby zostanie wywołany z opcją `-w`, będzie ostrzegał o wszystkich zmiennych lokalnych blokowych przesłaniających istniejące zmienne.

Oczywiście każdy blok może mieć więcej niż jeden parametr i jedną zmienną. Poniżej znajduje się blok z dwoma parametrami i trzema zmiennymi lokalnymi:

```
hash.each {|key,value; i,j,k| ... }
```

5.4.4. Przekazywanie argumentów do bloku

Napisaliśmy wcześniej, że parametry bloków w dużym stopniu przypominają parametry metod. Nie są jednak z nimi identyczne. Wartości argumentów znajdujące się za słowem kluczowym `yield` są przypisywane do parametrów blokowych zgodnie z zasadami, które są bliższe regułom przypisywania wartości zmiennym niż wywoływania metod. W związku z tym instrukcja `yield k, v` wykonywana przez iterator podczas wywoływania bloku zadeklarowanego z parametrami `|key, value|` jest równoznaczna z poniższą instrukcją przypisania:

```
key,value = k,v
```

Iterator `Hash.each_pair` przesyła parę klucz-wartość następująco⁶:

```
{:one=>1}.each_pair {|key,value| ... } #key=:one, value=1
```

W Ruby 1.8 jest nawet bardziej oczywiste, że wywołanie bloku używa przypisania do zmiennej. Przypomnijmy, że w Ruby 1.8 parametry mają zasięg lokalny blokowy tylko wówczas, gdy nie są używane jako zmienne lokalne metody nadrzednej. Jeśli są już zmiennymi lokalnymi, następuje tylko przypisanie im wartości. W rzeczywistości Ruby 1.8 pozwala na użycie dowolnego rodzaju zmiennej jako parametru blokowego, także zmiennych globalnych i obiektowych:

```
{:one=>1}.each_pair {|$key, @value| ... } # Nie działa w Ruby 1.9.
```

Iterator ustawia zmienną globalną `$key` na `:one`, a zmienną obiektową `@value` na `1`. Jak było już wspomniane, parametry bloków w Ruby 1.9 mają zasięg w obrębie swoich bloków. Oznacza to także, że parametry bloków nie mogą już być zmiennymi globalnymi ani obiektowymi.

Iterator `Hash.each` zamienia pary klucz-wartość na dwa elementy jednej tablicy. Bardzo często jednak spotyka się kod podobny do poniższego:

```
hash.each { |k,v| ... } # Klucz i wartość przypisane do parametrów k i v.
```

Można także zastosować przypisanie równolegle. Zwrócona dwuelementowa tablica jest przypisywana do zmiennych `k` i `v`:

```
k,v = [key, value]
```

⁶ Metoda `each_pair` w Ruby 1.8 przesyła do bloku dwie osobne wartości. W Ruby 1.9 metoda `each_pair` jest synonimem metody `each` i przekazuje pojedynczy argument tablicowy, co zostanie wyjaśnione wkrótce. Zaprezentowany tu kod działa poprawnie w obu wersjach języka.

Zgodnie z zasadami przypisania równoległego (zobacz podrozdział 4.5.5) pojedyncza tablica znajdująca się po prawej stronie zostaje rozbita na poszczególne elementy, które zostają przypisane do zmiennych znajdujących się po lewej stronie.

Wywoływanie bloku nie odbywa się dokładnie tak samo jak przypisanie równoległe. Wyobraź sobie iterator przekazujący dwie wartości do swojego bloku. Zgodnie z zasadami przypisania równoległego można oczekiwac możliwości deklaracji bloku z jednym parametrem i spowodowania, że te dwie wartości zostaną automatycznie wstawione do tablicy. Tak jednak nie jest:

```
def two; yield 1,2; end  # Iterator zwracający dwie wartości.  
two { |x| p x }        # Ruby 1.8: ostrzega i drukuje [1,2].  
two { |x| p x }        # Ruby 1.9: drukuje 1, nie ostrzega.  
two { |*x| p x }       # Obie wersje: drukuje [1,2]; bez ostrzeżenia.  
two { |x,| p x }        # Obie wersje: drukuje 1; bez ostrzeżenia.
```

W Ruby 1.8, jeśli zostanie zdefiniowany tylko jeden parametr blokowy i istnieje kilka argumentów `yield`, są one wstawiane do tablicy. Nie jest to jednak zalecane i powoduje wygenerowanie komunikatu ostrzegawczego. W Ruby 1.9 pierwsza wartość jest przypisywana do parametru blokowego, a druga po cichu odrzucana. Aby kilka zwróconych wartości zostało zapakowanych do tablicy i przypisanych do jednego parametru blokowego, konieczne jest postawienie przed tym parametrem znaku `*`, dokładnie tak samo jak w deklaracji metody (szczegółowy opis deklaracji i parametrów metod znajduje się w rozdziale 6.). Ponadto należy zauważyc, że istnieje możliwość jawnego odrzucenia drugiej zwrotnej wartości poprzez deklarację listy parametrów blokowych kończącej się przecinkiem, jakby mówiąc: „Jest jeszcze jeden parametr, ale jest on nieużywany i nie chce mi się wymyślać dla niego nazwy”.

Mimo iż w tym przypadku wywołanie bloku nie działa jak przypisanie równoległe, nie przypomina ono również wywołania metody. Jeśli metoda zostanie zadeklarowana z jednym argumentem, a następnie zostaną do niej przekazane dwa argumenty, Ruby nie wydrukuję ostrzeżenia, a zgłosi błąd.

W Ruby 1.8 tylko ostatni parametr blokowy może mieć prefiks `*`. W Ruby 1.9 usunięto to ograniczenie i dowolny jeden parametr blokowy, bez względu na swoje położenie na liście, może mieć prefiks `*`:

```
def five; yield 1,2,3,4,5; end      # Zwrot pięciu wartości.  
five do |head, *body, tail|         # Nadwyżkowe wartości przechodzą do tablicy body.  
  print head, body, tail           # Drukuję "1[2,3,4]5".  
end
```

Instrukcja `yield`, podobnie jak wywołania metod (zobacz podrozdział 4.6.4), pozwala na stosowanie jako wartości ostatniego argumentu goli tablic asocjacyjnych. To znaczy że jeśli ostatnim argumentem instrukcji `yield` jest literal tablicy asocjacyjnej, można pominąć jego nawiasy klamrowe. Ponieważ iteratory rzadko zwracają tablice asocjacyjne, oto zaimprowizowany przykład to obrazujący:

```
def hashiter; yield :a=>1, :b=>2; end  # Brak nawiasów klamrowych.  
hashiter { |hash| puts hash[:a] }          # Drukuję 1.
```

W Ruby 1.9 przed ostatnim parametrem blokowym można umieścić znak `&` wskazujący, że parametr ten ma otrzymać jakikolwiek blok związany z jego wywołaniem. Przypomnij sobie jednak, że z wywołaniem instrukcji `yield` nie może być związany żaden blok. W rozdziale 6. dowiesz się, że blok może zostać przekonwertowany na obiekt klasy `Proc` i że bloki **mogą** być związane z wywołaniami obiektów klasy `Proc`. Poniższy przykład kodu powinien nabrac sensu po przeczytaniu rozdziału 6.:

```
# Niniejszy obiekt klasy Proc spodziewa się bloku.  
printer = lambda { |&b| puts b.call } # Wydruk wartości zwrotnej przez b.  
printer.call { "hi" } # Przekazanie bloku do bloku!
```

Ważną różnicą pomiędzy parametrami blokowymi a parametrami metod jest to, że te pierwsze — w przeciwieństwie do drugich — nie mogą mieć wartości domyślnych. Oznacza to, że poniższy kod jest **niepoprawny**:

```
[1,2,3].each { |x,y=10| print x*y } # Błąd składni!
```

W Ruby 1.9 zdefiniowano nową składnię służącą do tworzenia obiektów klasy `Proc`, która pozwala, aby argumenty miały wartości domyślne. Szczegóły na ten temat znajdują się w rozdziale 6., ale powyższy kod można przepisać następująco:

```
[1,2,3].each &->(x,y=10) { print x*y } # Drukuje "102030".
```

5.5. Kontrola przepływu sterowania

Poza instrukcjami warunkowymi, pętlami i iteratorami Ruby obsługuje także kilka instrukcji, które pozwalają zmienić przepływ sterowania w programie. Instrukcje te to:

`return`

Powoduje wyjście z metody i zwrot wartości do kodu wywołującego.

`break`

Powoduje wyjście z pętli (lub iteratora).

`next`

Przerywa bieżącą iterację pętli (lub iteratora) i powoduje przejście do kolejnej iteracji.

`redo`

Uruchamia pętlę lub iterator od początku.

`retry`

Ponownie uruchamia iterator, jeszcze raz obliczając wartość całego wyrażenia. Słowo kluczowe `retry` może też być używane w obsłudze wyjątków, o czym piszemy dalej.

`throw/catch`

Bardzo ogólna struktura sterująca, która nazwą i działaniem przypomina mechanizm propagacji i obsługi wyjątków. Słowa kluczowe `throw` i `catch` nie są głównym mechanizmem obsługi wyjątków w języku Ruby (tę rolę pełnią słowa kluczowe `raise` i `rescue` opisane dalej w tym rozdziale). W zamian są używane jako pewien rodzaj wielopoziomowej lub etykietowanej instrukcji `break`.

Kolejne podrozdziały szczegółowo opisują każdą z wymienionych powyżej instrukcji.

5.5.1. Instrukcja `return`

Instrukcja `return` powoduje zwrocenie przez otaczającą metodę wartości do kodu wywołującego. Osoby znające język C, Java albo inny do nich podobny prawdopodobnie intuicyjnie rozumieją działanie instrukcji `return`. Mimo to rozdział ten trzeba przeczytać, ponieważ działanie tej instrukcji w blokach może nie być już takie intuicyjne.

Po instrukcji `return` można wstawić opcjonalne wyrażenie lub listę wyrażeń rozdzielonych przecinkami. Jeśli nie ma żadnego wyrażenia, wartością zwrotną metody jest `nil`. Gdy jest jedno wyrażenie, wartością zwrotną metody jest wartość zwrotna tego wyrażenia. Jeżeli po

słowie kluczowym znajduje się więcej niż jedno wyrażenie, wartością zwrotną jest tablica wartości tych wyrażeń.

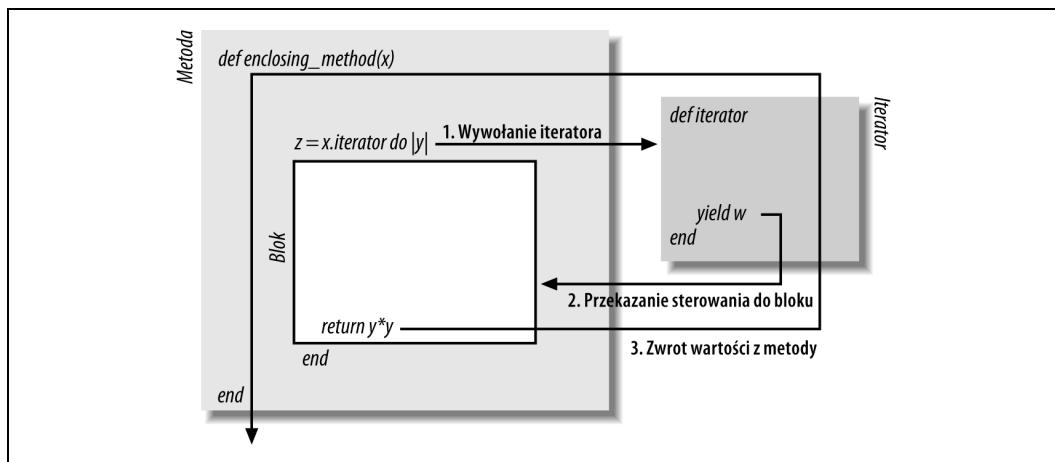
Warto zauważyć, że większość metod nie wymaga instrukcji `return`. Kiedy sterowanie do trze do końca metody, ta automatycznie zwraca wartość do kodu wywołującego. Wartością zwrotną w takim przypadku jest wartość ostatniego wyrażenia w metodzie. Większość programistów Ruby pomija instrukcję `return`, gdy ta nie jest wymagana. Zamiast w ostatnim wierszu metody pisać `return x`, piszą po prostu `x`.

Instrukcja `return` jest przydatna, jeśli ktoś chce zwrócić wartość z metody przedwcześnie lub chce zwrócić kilka wartości. Na przykład:

```
# Zwrot dwóch kopii x, jeżeli x nie ma wartości nil.  
def double(x)  
  return nil if x == nil    # Przedwczesny zwrot wartości.  
  return x, x.dup           # Zwrot kilku wartości.  
end
```

Przy pierwszym zetknięciu z blokami w języku Ruby naturalne jest wyobrażanie sobie ich jako pewnego rodzaju zagnieżdżonych funkcji lub minutek. Taki sposób rozumowania może prowadzić do konkluzji, że instrukcja `return` powoduje zwrócenie wartości przez blok do iteratora, który przekazał do niego sterowanie. Jednak bloki to nie metody i instrukcja `return` nie działa w nich w ten sposób. W rzeczywistości instrukcja ta jest bardzo konsekwentna — zawsze powoduje zwrot przez metodę nadziedzoną, bez względu na to, jak głęboko jest zagnieżdżona w blokach⁷.

Należy zauważyć, że metoda nadziedzona nie jest tym samym co metoda wywołująca. Kiedy instrukcja `return` jest użyta w bloku, nie powoduje ona tylko zwrotu wartości przez blok. Nie powoduje również zwrotu wartości przez iterator, który ten blok wywołał. Instrukcja `return` zawsze zmusza do zwrotu wartości metodę nadziedzoną. Metoda nadziedzona (czasami nazywana metodą nadziedzoną leksykalnie) to metoda, w której znajduje się blok. Rysunek 5.2 ilustruje działanie instrukcji `return` w bloku.



Rysunek 5.2. Instrukcja `return` w bloku

⁷ Wyjątek od tej zasady przedstawiamy przy opisie obiektów lambda w podrozdziale 6.5.5.1. Lambda to rodzaj funkcji utworzonej z bloku. Działanie instrukcji `return` w lambdach różni się od jej działania w zwykłych blokach.

Poniższy kod definiuje metodę, która za pomocą instrukcji `return` zwraca wartość z wnętrza bloku:

```
# Zwrot indeksu pierwszego wystąpienia celu w tablicy lub wartości nil.  
# Zauważ, że niniejszy kod powiela tylko metodę Array#index.  
def find(array, target)  
  array.each_with_index do |element, index|  
    return index if (element == target) # Zwrot wartości przez metodę find.  
  end  
  nil # Jeśli element nie zostanie znaleziony, zwracana jest wartość nil.  
end
```

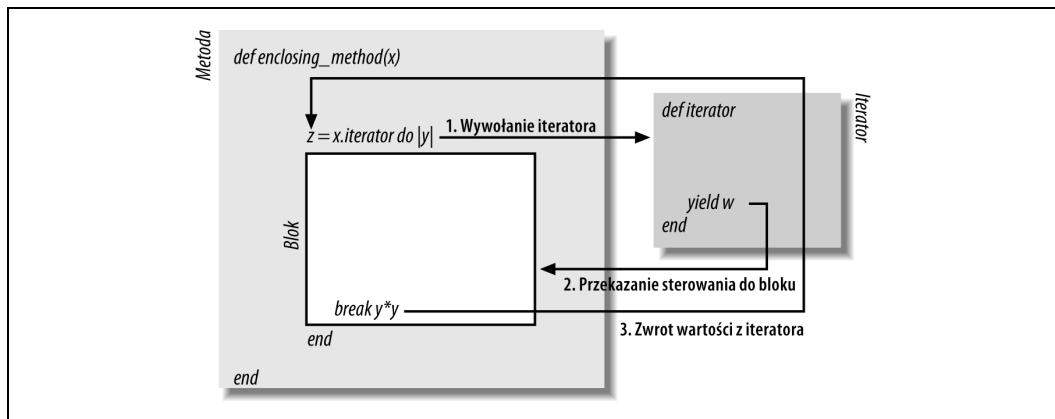
Instrukcja `return` w powyższej procedurze nie powoduje zwrotu wartości przez blok do iteratora, który go wywołał, ani zwrotu wartości przez iteratator `each_with_index`. Powoduje natomiast, że metoda `find` zwraca wartość do kodu, który ją wywołał.

5.5.2. Instrukcja `break`

Instrukcja `break` użyta w pętli przekazuje sterowanie do pierwszego wyrażenia znajdującego się za nią. Programiści języka C, Javy lub innego do nich podobnego znają ten sposób używania instrukcji `break` w pętlach:

```
while(line = gets.chomp)      # Początek pętli.  
  break if line == "quit"      # Jeśli ta instrukcja break zostanie wykonana...  
  puts eval(line)  
end  
puts "Do widzenia"          # ... sterowanie zostanie przekazane do tego miejsca.
```

Jak widać, użycie instrukcji `break` wewnętrzny bloku jest z leksykalnego punktu widzenia tym samym co użycie jej w pętli. Jednak z punktu widzenia stosu wywołań instrukcja `break` w bloku jest bardziej skomplikowana, ponieważ zmusza metodę iteracyjną, z którą związany jest blok, do zwrotu wartości. Użycie jej w jakimkolwiek innym miejscu powoduje błąd `LocalJumpError`.



Rysunek 5.3. Instrukcja `break` w bloku

5.5.2.1. Instrukcja `break` z wartością

Przypomnijmy, że wszystkie konstrukcje syntaktyczne w języku Ruby są wyrażeniami i każda z nich może mieć wartość. Instrukcja `break` może określać wartość dla pętli lub iteratora, z którego powoduje wyjście. Po słowie kluczowym `break` można wstawić wyrażenie lub listę

wyrażeń oddzielonych przecinkami. Jeżeli instrukcja `break` zostanie zastosowana bez żadnego wyrażenia, wartością zwrotną pętli lub metody iteracyjnej będzie `nil`. Jeśli ma ona tylko jedno wyrażenie, jego wartość staje się wartością pętli lub iteratora. Gdy natomiast instrukcja `break` ma kilka wyrażeń, ich wartości zwrotne są wstawiane do tablicy, a ta staje się wartością pętli lub iteratora.

Dla kontrastu — pętla `while`, która kończy się w sposób naturalny bez instrukcji `break`, ma zawsze wartość zwrotną `nil`. Wartość zwrotna iteratora kończącego działanie w normalny sposób jest zdefiniowana w metodzie iteracyjnej. Wiele iteratorów, jak `times` i `each`, zwraca obiekt, na rzecz którego zostały wywołane.

5.5.3. Instrukcja `next`

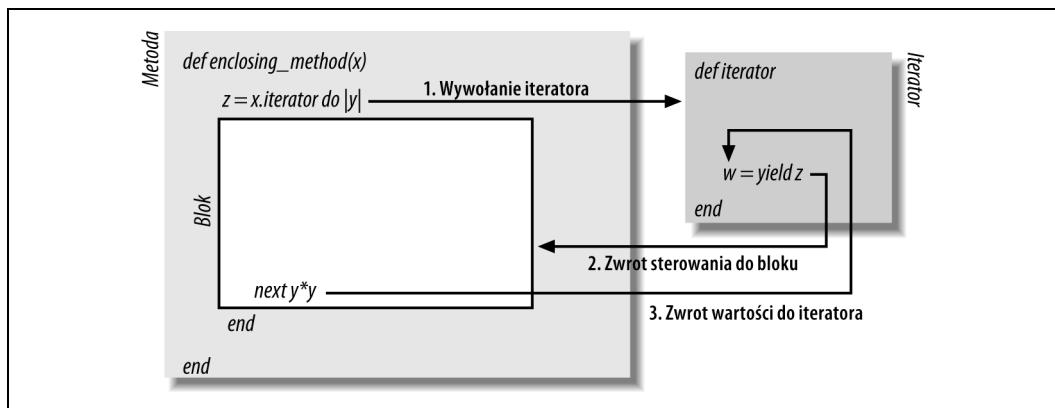
Instrukcja `next` powoduje przerwanie bieżącej iteracji pętli lub iteratora i rozpoczęcie kolejnej. Programistom języków C i Java ta struktura jest znana pod nazwą `continue`. Oto przykład użycia jej w pętli:

```
while(line = gets.chomp)      # Początek pętli.  
  next if line[0,1] == "#"    # Jeżeli ten wiersz jest komentarzem, przejdź do kolejnego.  
  puts eval(line)  
  # Sterowanie przechodzi do tego miejsca po wykonaniu instrukcji next.  
end
```

Kiedy instrukcja `next` jest użyta w bloku, powoduje natychmiastowe wyjście z niego. Sterowanie jest wtedy zwracane do metody iteracyjnej, która może rozpocząć nową iterację, ponownie wywołując blok:

```
f.each do |line|  
  next if line[0,1] == "#"    # Iteracja przez wiersze pliku f.  
  puts eval(line)  
  # Sterowanie przechodzi do tego miejsca po wykonaniu instrukcji next.  
end
```

Użycie instrukcji `next` w bloku jest pod względem leksykalnym takie samo jak w pętlach `while`, `until` i `for-in`. Jednak z punktu widzenia kolejności wywołań przypadek z blokiem jest bardziej skomplikowany, co zilustrowano na rysunku 5.4.



Rysunek 5.4. Instrukcja `next` w bloku

next, break i return

Warto porównać rysunek 5.4 z rysunkami 5.2 i 5.3. Instrukcja `next` powoduje zwrot wartości przez blok do metody iteracyjnej, która go wywołała. Instrukcja `break` powoduje zwrot wartości przez blok do swojego iteratora, a przez iterator do metody nadrzędnej. Natomiast instrukcja `return` powoduje zwrot przez blok wartości do iteratora, przez iterator do metody nadrzędnej, a przez nią do kodu wywołującego.

Instrukcji `next` można używać wyłącznie w pętlach i blokach. Jej użycie w jakimkolwiek innym miejscu powoduje błąd `LocalJumpError`.

5.5.3.1. Instrukcja next a wartość bloku

Podobnie jak słowa kluczowe `return` i `break`, słowo kluczowe `next` może występować samodzielnie lub z jednym albo większą liczbą wyrażeń oddzielonych przecinkami. Kiedy instrukcja `next` jest używana w pętli, wszystkie wartości znajdujące się za nią są ignorowane. Natomiast w bloku wyrażenie lub wyrażenia te stają się wartością zwrotną instrukcji `yield`, która wywołała blok. Jeśli po słowie kluczowym `next` nie ma żadnego wyrażenia, wartością instrukcji `yield` jest `nil`. W przypadku gdy po `next` znajduje się jedno wyrażenie, wartość tego wyrażenia jest wartością zwrotną instrukcji `yield`. Jeżeli natomiast po słowie kluczowym `next` znajduje się kilka wyrażeń, wartością zwrotną instrukcji `yield` jest tablica wartości zwróconych przez te wyrażenia.

Przy omawianiu instrukcji `return` wyjaśnialiśmy, że bloki nie są funkcjami i że instrukcja `return` nie zmusza bloku do zwrotu wartości do iteratora, który ten blok wywołał. Jak widać, instrukcja `next` właśnie to robi. Oto przykładowy kod demonstrujący sytuację, w której można jej użyć w ten sposób:

```
squareroots = data.collect do |x|
  next 0 if x < 0 # Zwraca 0 dla ujemnych wartości.
  Math.sqrt(x)
end
```

Normalnie wartością wyrażenia `yield` jest wartość ostatniego wyrażenia w bloku. Podobnie jak w przypadku instrukcji `return`, często nie ma potrzeby jawnego określania wartości za pomocą `next`. Powyższy kod można równie dobrze zapisać następująco:

```
squareroots = data.collect do |x|
  if (x < 0) then 0 else Math.sqrt(x) end
end
```

5.5.4. Instrukcja redo

Instrukcja `redo` ponownie uruchamia bieżącą iterację pętli lub iteratora. To nie to samo co instrukcja `next`, która przenosi sterowanie na koniec pętli lub bloku, dzięki czemu może zacząć się nowa iteracja. Instrukcja `redo` przenosi sterowanie z powrotem na początek pętli lub bloku, dzięki czemu ta sama iteracja może zacząć się jeszcze raz. Instrukcja ta jest nowością dla programistów znających dotychczas języki podobne do C.

`redo` przenosi sterowanie do pierwszego wyrażenia w ciele pętli lub bloku. Nie sprawdza ponownie warunku pętli i nie pobiera kolejnego elementu z iteratora. Poniższa pętla `while` zakonczyłaby się normalnie po trzech iteracjach, ale przez instrukcję `redo` wykonuje cztery iteracje:

```

i = 0
while(i < 3)    # Drukuję "0123" zamiast "012".
  # Po wykonaniu instrukcji redo sterowanie wraca do tego miejsca.
  print i
  i += 1
  redo if i == 3
end

```

Instrukcja `redo` nie jest powszechnie używana, przez co wiele przykładów jej użycia, jak powyższy, zostało wymyślonych. Jednym z jej zastosowań jest wychodzenie z błędów wejścia powstających przy pobieraniu danych od użytkownika. W poniższym kodzie instrukcja `redo` została użyta w bloku do tego właśnie celu:

```

puts "Podaj pierwsze słowo, które przychodzi ci do głowy"
words = %w(jabłko banan wiśnia)  # Skrócony zapis ["jabłko", "banan", "wiśnia"].
response = words.collect do |word|
  # Po wykonaniu instrukcji redo sterowanie wraca do tego miejsca.
  print word + ">"           # Zapytanie użytkownika.
  response = gets.chomp        # Pobranie odpowiedzi.
  if response.size == 0
    word.upcase!               # Jeśli użytkownik nic nie wpisał,
    redo                      # położ nacisk na prośbę, drukując ją wielkimi literami.
  else                        # Przejście na początek bloku.
    response                   # Zwrócenie odpowiedzi.
  end
end

```

5.5.5. Instrukcja `retry`

Instrukcja `retry` jest zazwyczaj używana w klauzuli `rescue`. Jej przeznaczeniem jest ponowne wykonanie bloku kodu, który spowodował wyjątek. Informacje na ten temat znajdują się w podrozdziale 5.6.3.5. Jednak w Ruby 1.8 instrukcja ta ma inne zastosowanie — uruchamia iterację iteratora (lub jakiekolwiek wywołanie metody) od początku. Instrukcja `retry` w tej roli jest używana niezwykle rzadko i w Ruby 1.9 postanowiono ją usunąć. Dlatego też należy ją uznać za wycofywaną własność języka i nie używać jej w nowych programach.

W bloku instrukcja `retry` nie tylko ponownie wykonuje bieżące wywołanie bloku. Zmusza też blok i metodę iteracyjną do zakończenia działania, a następnie ponownie uruchamia iterację, wcześniej jeszcze raz obliczając wyrażenie iteratora. Spójrz na poniższy fragment programu:

```

n = 10
n.times do |x|    # Iteracja n razy od 0 do n-1.
  print x          # Drukowanie numeru iteracji.
  if x == 9        # Jeśli doszedłeś do numeru 9,
    n -= 1          # zmniejsz n (następny razem nie dojdziesz do 9!).
    retry           # Restart iteracji.
  end
end

```

W powyższej procedurze zrestartowano iterator za pomocą instrukcji `retry`, ale zachowano ostrożność, aby uniknąć powstania nieskończonej pętli. Przy pierwszym wywołaniu drukowana jest liczba 0123456789, po czym następuje ponowne uruchomienie iteracji. Przy drugim wywołaniu drukowana jest liczba 012345678 i nie następuje ponowne uruchomienie.

Tajemnica instrukcji `retry` polega na tym, że nie uruchamia ona iteratora za każdym razem tak samo. Ponownie oblicza jego wyrażenie, co oznacza, że argumenty tego iteratora (a nawet obiekt, na rzecz którego jest on wywoływany) mogą być inne przy każdym jego uruchomieniu. Tego typu ponowne obliczanie wyrażeń może być trudne do zrozumienia dla osób nieprzyzwyczajonych do takich bardzo dynamicznych języków programowania jak Ruby.

Zastosowanie instrukcji `retry` nie ogranicza się do bloków. Działa ona w ten sposób, że zawsze ponownie oblicza wywołanie najbliższej metody nadrzednej. Oznacza to, że można jej używać (przed Ruby 1.9) do pisania iteratorów podobnych do poniższego, który działa jak pętla `while`:

```
# Niniejsza metoda działa jak pętla while: jeśli x nie ma wartości nil ani false,  
# następuje wywołanie bloku i ponowna próba zrestartowania pętli oraz sprawdzenie  
# warunku. Metoda ta nieco różni się od prawdziwej pętli while:  
# ciało pętli można umieścić, podobnie jak w języku C, w nawiasach klamrowych.  
# Dodatkowo zmienne używane tylko w obrębie ciała pętli pozostają lokalne w tym bloku.  
def repeat_while(x)  
  if x      # Jeśli warunek nie zwrócił wartości nil ani false,  
    yield   # następuje uruchomienie ciała pętli.  
    retry   # Ponowne wypróbowanie i obliczenie warunku pętli.  
  end  
end
```

5.5.6. Metody `throw` i `catch`

Metody `throw` i `catch` zostały zdefiniowane w module `Kernel`. Tworzą one strukturę sterującą, która działaniem przypomina wielopoziomową instrukcję `break`. Instrukcja `throw` nie przerywa tylko aktualnej pętli lub aktualnego bloku, ale pozwala na przejście o dowolną liczbę poziomów do góry, powodując zakończenie działania bloku zdefiniowanego w `catch`. Metoda `catch` nie musi nawet znajdować się w tej samej metodzie co `throw`. Może być w metodzie wywołującej, a nawet jeszcze wyżej na stosie wywołań.

W językach takich jak Java i JavaScript pętle można etykietować za pomocą dowolnych prefiksów. W takim przypadku struktura sterująca zwana instrukcją `break` z etykietą powoduje zakończenie działania pętli o podanej nazwie. W Ruby blok kodu z etykietą jest definiowany przez metodę `catch`, a metoda `throw` zmusza go do zakończenia działania. Jednak metody `throw` i `catch` są bardziej ogólne od instrukcji `break` z etykietą. Między innymi można ich używać w połączeniu z dowolnego rodzaju instrukcjami, ponieważ nie są one ograniczone do pętli. Dodatkowo metoda `throw` może przechodzić w górę stosu i powodować zakończenie działania bloku w metodzie wywołującej.

Programiści znający języki Java i JavaScript znają `throw` i `catch` jako słowa kluczowe służące do zgłoszania i obsługi wyjątków. W języku Ruby obsługa wyjątków wygląda inaczej. Używa się do tego słów kluczowych `raise` i `rescue`, o których piszemy dalej w tym rozdziale. Jednak to podobieństwo do mechanizmu obsługi wyjątków jest zamierzone. Wywołanie metody `throw` w znacznym stopniu przypomina zgłoszanie wyjątku. Ponadto sposób jej propagacji przez zakres leksykalny, a następnie w górę stosu wywołań, w dużym stopniu przypomina sposób propagacji wyjątków na zewnątrz i do góry (dużo więcej na temat propagacji wyjątków piszemy dalej). Mimo podobieństwa do wyjątków najlepiej jest traktować metody `throw` i `catch` jako strukturę sterującą ogólnego przeznaczenia (jeśli nie są zbyt często używane) niż jako mechanizm obsługi wyjątków. Aby zasygnalizować błąd lub wyjątkową sytuację, należy zamiast `throw` użyć `raise`.

Poniższy kod przedstawia sposób użycia metod `throw` i `catch` do wyzwalania się z zagnieżdzonych pętli:

```
for matrix in data do  
  catch :missing_data do          # Przetwarzanie głęboko zagnieżdzonej struktury danych.  
    for row in matrix do          # Nadanie instrukcji etykiety, aby można było z niej wyjść.  
      for value in row do  
        throw :missing_data unless value # Wyjście z dwóch pętli naraz.  
        # W przeciwnym wypadku przetwarzane są dane.
```

```
end
end
end
# Koniec następuje w tym miejscu, po zakończeniu przetwarzania każdej macierzy przez pętle.
# Docierasz tutaj także wtedy, gdy zostanie spowodowany wyjątek :missing_data.
```

Należy zauważyć, że metoda `catch` przyjmuje argument w postaci symbolu oraz blok. Wykonuje ona blok i zwraca wartość, kiedy blok zakończy działanie lub zostanie wyrzucony określony symbol. Metoda `throw` również przyjmuje symbol jako argument i zmusza odpowiednie wywołanie metody `catch` do zwrócenia wartości. Jeśli żadne wywołanie metody `catch` nie pasuje do symbolu przekazanego do metody `throw`, zgłoszany jest wyjątek `NameError`. Zarówno metodę `catch`, jak i `throw` można wywołać przy użyciu łańcuchów jako argumentów zamiast symboli. Zostaną one wewnętrznie przekonwertowane na symbole.

Jedną z ciekawych cech metod `throw` i `catch` jest to, że działają one nawet wtedy, gdy znajdują się w różnych metodach. Można by zmodyfikować zaprezentowany kod, wstawiając najgdyś zagnieźdzoną pętlę do osobnej metody, a przepływ sterowania działałby nadal bez zarzutu.

Jeśli metoda `throw` nigdy nie zostaje wywołana, metoda `catch` zwraca wartość ostatniego wyrażenia w swoim bloku. W sytuacji gdy metoda `throw` zostaje wywołana, odpowiadającej jej metoda `catch` zwraca domyślną wartość `nil`. Można jednak określić inną wartość zwrotną dla metody `catch`, przekazując do metody `throw` drugi argument. Wartość zwrotna metody `catch` pomaga odróżnić normalne zakończenie działania bloku kodu od nienormalnego zakończenia przy użyciu metody `throw`. Dzięki temu można napisać kod wykonujący wszelkie dodatkowe działania wymagane w odpowiedzi na metodę `throw`.

Metody `throw` i `catch` nie są często używane. Jeżeli znajdziesz w jednej metodzie użyte obie te metody, spróbuj zrefaktoryzować `catch` do osobnej definicji metody, a `throw` zastąpić instrukcją `return`.

5.6. Wyjątki i ich obsługa

Wyjątek to obiekt reprezentujący jakiś rodzaj wyjątkowej sytuacji. Oznacza on, że coś poszło źle. Może to być błąd programistyczny — próba dzielenia przez zero, wywołanie metody na rzecz obiektu, który jej nie udostępnia, lub przekazanie do metody nieprawidłowego argumentu. Może to też być wynik jakiejś sytuacji zewnętrznej — wysłanie żądania sieciowego, kiedy sieć nie działa, lub próba utworzenia obiektu przy braku pamięci w systemie.

Kiedy nastąpi jedna z powyższych sytuacji, **zgłoszany** (lub **wyrzucany**) jest wyjątek. Domyślnie programy Ruby kończą wtedy działanie. Można jednak zadeklarować procedury obsługi wyjątków. Procedura taka jest blokiem kodu wykonywanym w chwili wystąpienia wyjątku podczas wykonywania innego bloku kodu. W tym sensie wyjątki są rodzajem instrukcji sterującej. Zgłoszenie wyjątku powoduje przekazanie sterowania do kodu obsługi tego wyjątku. Jest to podobne do użycia instrukcji `break` do wychodzenia z pętli. Jak się jednak niebawem przekonasz, wyjątki znacznie różnią się od instrukcji `break`. Mogą przekazywać sterowanie na zewnątrz wielu otaczających bloków, a nawet w góre stosu wywołań, aby dotrzeć do procedury obsługi.

Do zgłoszania wyjątków służy metoda `raise` z modułu `Kernel`. Kod obsługujący wyjątki przechowywany jest przez klauzulę `rescue`. Wyjątki zgłoszone przez metodę `raise` są egzemplarzami klas `Exception` lub jednej z jej podklas. Opisane wcześniej metody `throw` i `catch`

nie są przeznaczone do sygnalizowania i obsługi wyjątków, ale symbol wyrzucony przez metodę `throw` jest propagowany w taki sam sposób jak wyjątek zgłoszony przez metodę `raise`. Obiekty wyjątków, ich propagacja, metoda `raise` oraz klauzula `rescue` zostały szczegółowo opisane w kolejnych podrozdziałach.

5.6.1. Klasy i obiekty wyjątków

Obiekty wyjątków są egzemplarzami klasy `Exception` lub jednej z jej wielu podklas. Podklasy te z reguły nie definiują żadnych nowych metod lub zachowań, ale pozwalają na podział wyjątków na typy. Hierarchia tych klas została przedstawiona na rysunku 5.5.

```
Object
  +-+Exception
    +-+NoMemoryError
    +-+ScriptError
    |   +-+LoadError
    |   +-+NotImplementedError
    |   +-+SyntaxError
    +-+SecurityError      # StandardError w 1.8.
    +-+SignalException
    |   +-+Interrupt
    +-+SystemExit
    +-+SystemStackError     # StandardError w 1.8.
    +-+StandardError
      +-+ArgumentError
      +-+FiberError        # Nowość w 1.9.
      +-+IOError
      |   +-+EOFError
      +-+IndexError
      |   +-+KeyError       # Nowość w 1.9.
      |   +-+StopIteration  # Nowość w 1.9.
      +-+LocalJumpError
      +-+NameError
      |   +-+NoMethodError
      +-+RangeError
      |   +-+FloatDomainError
      +-+RegexpError
      +-+RuntimeError
      +-+SystemCallError
      +-+ThreadError
      +-+TypeError
      +-+ZeroDivisionError
```

Rysunek 5.5. Hierarchia klas wyjątków

Nie trzeba znać wszystkich klas. Ich nazwy mówią same za siebie. Ważne jest, aby zauważyc, że większość podklas rozszerza klasę o nazwie `StandardError`. Są to normalne wyjątki, które w typowym programie Ruby są obsługiwane. Pozostałe klasy reprezentują wyjątki niższego poziomu, poważniejsze lub bardziej skomplikowane do naprawienia. Typowe programy Ruby nie próbują ich obsługiwać.

Używając narzędzia `ri` do znalezienia dokumentacji tych klas, można odkryć, że większość z nich nie posiada żadnej dokumentacji. Częściowo jest to spowodowane tym, że gros z nich nie dodaje żadnych nowych metod do tych, które zostały zdefiniowane w klasie `Exception`. Ważne jest, aby wiedzieć, kiedy można zgłosić wyjątek danej klasy. Te informacje można zazwyczaj znaleźć w metodach zgłaszających wyjątki, a nie w samych klasach tych wyjątków.

5.6.1.1. Metody obiektów wyjątków

Klasa `Exception` definiuje dwie metody zwracające szczegółowe informacje na temat wyjątków. Metoda `message` zwraca łańcuch, który może zawierać możliwe do odczytu przez człowieka informacje na temat, co poszło nie tak. Jeśli program Ruby zakończy się przez nieobsłużony wyjątek, użytkownikowi końcowemu jest zazwyczaj prezentowana właśnie ta informacja. Jednak jej głównym przeznaczeniem jest wspomaganie programisty w zdiagnozowaniu problemu.

Drugą ważną metodą obiektów wyjątków jest `backtrace`. Zwraca ona tablicę łańcuchów reprezentującą stos wywołań, pokazującą, jak wyglądał on w chwili zgłoszenia wyjątku. Każdy element tablicy jest łańcuchem w następującym formacie:

```
nazwapliku : numerwiersza in nazwametody
```

Pierwszy element tablicy określa miejsce, gdzie został zgłoszony wyjątek, drugi — miejsce wywołania metody, która go spowodowała, trzeci — miejsce wywołania tamtej metody itd. (metoda `caller` z modułu `Kernel` zwraca dane ze śledzenia stosu w takim samym formacie, można ją wypróbować w `irb`). Obiekty wyjątków są z reguły tworzone przez metodę `raise`. Po utworzeniu obiektu metoda ta odpowiednio ustawia informacje o stosie wyjątku. Tworząc własny obiekt wyjątku, można stos ustawić w dowolny sposób za pomocą metody `set_backtrace`.

5.6.1.2. Tworzenie obiektów wyjątków

Obiekty wyjątków są z reguły tworzone przez metodę `raise`, o czym przekonasz się dalej. Można też tworzyć je za pomocą metody `new` oraz innej metody klasowej o nazwie `exception`. Obie przyjmują jeden opcjonalny argument łańcuchowy. Jeśli zostanie on podany, łańcuch staje się wartością metody `message`.

5.6.1.3. Definiowanie nowych klas wyjątków

Przy definiowaniu modułu Ruby często właściwe jest zdefiniowanie własnej podklasy klasy `StandardError` dla wyjątków charakterystycznych dla tego modułu. Może to być nawet bardzo prosta mieszcząca się w jednym wierszu klasa:

```
class MyError < StandardError; end
```

5.6.2. Zgłaszanie wyjątków za pomocą metody `raise`

Metoda `raise` z modułu `Kernel` zgłasza wyjątek. Jej synonimem `fail` używa się wówczas, gdy zachodzi podejrzenie, że zgłoszony wyjątek spowoduje zamknięcie programu. Jest kilka sposobów na wywołanie metody `raise`:

- Jeśli metoda `raise` zostanie wywołana bez żadnych argumentów, utworzy nowy obiekt `RuntimeError` (bez żadnego komunikatu) i zgłosi go. Gdy natomiast metoda ta zostanie użyta bez żadnych argumentów w klauzuli `rescue`, ponownie zgłosi wyjątek, który był obsługiwany.
- Jeżeli metoda `raise` zostanie wywołana z jednym obiektem klasy `Exception` jako argumentem, zgłosi podany wyjątek. Mimo prostoty nie jest to często spotykany sposób używania tej metody.

- W przypadku gdy metoda `raise` zostanie wywołana z jednym argumentem łańcuchowym, utworzy nowy obiekt `RuntimeError` z podanym łańcuchem jako komunikatem i zgłosi ten wyjątek. Jest to bardzo często spotykany sposób jej użycia.
- Jeśli pierwszym argumentem metody `raise` jest obiekt udostępniający metodę `exception`, `raise` wywołuje `exception` i zgłasza obiekt klasy `Exception` zwrócony przez metodę `exception`. Klasa `Exception` definiuje metodę `exception`, a zatem obiekt tej klasy można określić dla dowolnego wyjątku jako pierwszy argument metody `raise`.

Metoda `raise` przyjmuje łańcuch jako drugi opcjonalny argument. Jeżeli łańcuch ten jest podany, zostaje on przekazany do metody `exception` pierwszego argumentu. łańcuch ma służyć jako komunikat związany z wyjątkiem.

Metoda `raise` przyjmuje także opcjonalny trzeci argument. Może to być tablica łańcuchów, które zostaną zastosowane jako dane ze śledzenia stosu dla obiektu wyjątku. W sytuacji gdy argument ten nie zostanie podany, metoda `raise` ustawi dane stosu dla wyjątku samodzielnie (przy użyciu metody `caller` z modułu `Kernel`).

Poniższy kod definiuje prostą metodę zgłaszającą wyjątek, jeśli zostanie ona wywołana z parametrem o nieprawidłowej wartości:

```
def factorial(n)          # Definicja metody factorial z argumentem n.
    raise "zły argument" if n < 1  # Zgłoszenie wyjątku dla złej wartości n.
    return 1 if n == 1            # factorial(1) wynosi 1.
    n * factorial(n-1)          # Rekurencyjne obliczenie innych silni (factorial).
end
```

Niniejsza metoda wywołuje metodę `raise` z jednym argumentem łańcuchowym. Poniżej znajduje się kilka równoważnych sposobów zgłoszenia tego samego wyjątku:

```
raise RuntimeError, "zły argument" if n < 1
raise RuntimeError.new("zły argument") if n < 1
raise RuntimeError.exception("zły argument") if n < 1
```

W tym przypadku prawdopodobnie lepszy jest wyjątek klasy `ArgumentError` niż `RuntimeError`:

```
raise ArgumentError if n < 1
```

Pomocny byłby też bardziej szczegółowy komunikat:

```
raise ArgumentError, "Spodziewany argument >= 1. Otrzymano #{n}" if n < 1
```

Celem zgłoszanego tutaj wyjątku jest zasygnalizowanie problemu z wywołaniem metody `factorial`, nie z kodem wewnętrznie. Pierwszy element danych stosu zgłoszonego tutaj wyjątku będzie zawierał informację o miejscu wywołania metody `raise`. Drugi element tej tablicy będzie identyfikował kod, który wywołał metodę `factorial` ze złym argumentem. Aby bezpośrednio wskazać problematyczny fragment kodu, można dostarczyć własne dane stosu jako trzeci argument metody `raise` za pomocą metody `caller` z modułu `Kernel`:

```
if n < 1
  raise ArgumentError, "Spodziewany argument >= 1. Otrzymano #{n}", caller
end
```

Należy zauważyc, że metoda `factorial` sprawdza, czy jej argument mieści się we właściwym przedziale, ale nie kontroluje, czy jest on odpowiedniego typu. Sprawdzanie błędów można usprawnić w pierwszym wierszu metody, wstawiając poniższy kod:

```
raise TypeError, "Oczekiwana liczba całkowita" if not n.is_a? Integer
```

Z drugiej strony warto zobaczyć, co się stanie, gdy do pierwotnej wersji metody factorial zostanie podany argument łańcuchowy. Ruby porównuje argument n z liczbą całkowitą 1 za pomocą operatora <. Jeśli argument ten jest łańcuchem, porównywanie to nie ma sensu, a więc kończy się niepowodzeniem i zgłoszeniem wyjątku TypeError. Jeżeli argument jest egzemplarzem jakiejś klasy, która nie definiuje operatora <, zostaje zgłoszony wyjątek NoMethodError.

Należy zaznaczyć, że wyjątki mogą pojawiać się nawet wówczas, gdy metoda raise nie zostanie wywołana przez programistę. Dlatego ważne jest, aby umieć obsługiwać wyjątki, nawet jeśli nigdy nie zgłasza się ich własnoręcznie. Obsługa wyjątków została opisana w kolejnym podrozdziale.

5.6.3. Obsługa wyjątków przy użyciu klauzuli rescue

Metoda raise jest zdefiniowana w module Kernel. Natomiast klauzula rescue stanowi podstawowy składnik języka Ruby. Nie jest to samodzielna instrukcja, a klauzula, którą można dołączać do innych instrukcji. Najczęściej spotyka się ją w połączeniu z instrukcją begin; służy ona do oddzielania bloków kodu, w których mają być obsługiwane wyjątki. Instrukcja begin z klauzulą rescue wygląda następująco:

```
begin
  # W tym miejscu może znajdować się dowolna liczba instrukcji.
  # Zazwyczaj są one wykonywane bez żadnych problemów i
  # wykonywanie jest kontynuowane od miejsca za instrukcją end.
rescue
  # To jest klauzula rescue, w której znajduje się kod obsługi wyjątków.
  # Jeśli powyższy kod zgłosi wyjątek lub zostanie on przesłany z
  # jednej z metod wywołanych wyżej, wykonywany jest ten kod.
end
```

5.6.3.1. Nadawanie nazwy obiektowi wyjątku

W klauzuli rescue zmienna globalna \$! odwołuje się do obsługiwanej obiektu klasy Exception. Wykrywnik jest mnemonikiem — wyjątek to pewien rodzaj wykrywienia. Jeżeli program zawiera poniższy wiersz kodu:

```
require 'English'
```

można w zamian użyć zmiennej globalnej \$ERROR_INFO.

Lepszą alternatywą dla zmiennej \$! i \$ERROR_INFO jest zmienna dla wyjątku w samej klauzuli rescue:

```
rescue => ex
```

Instrukcje klauzuli rescue mogą teraz odwoływać się do obiektu klasy Exception reprezentującego wyjątek za pomocą zmiennej ex. Na przykład:

```
begin
  x = factorial(-1)
rescue => ex
  puts "#{ex.class}: #{ex.message}"  # Obsługa wyjątku polegająca na wydrukowaniu komunikatu.
end
```

Warto zauważyc, że klauzula rescue nie definiuje nowego zakresu dla zmiennych, a więc zmiana w niej zdefiniowana jest widoczna także poza nią. W przypadku gdy w klauzuli rescue zostanie użyta zmienna, obiekt wyjątku może być widoczny po jej zakończeniu, nawet jeśli zmienna \$! nie jest już ustaliona.

5.6.3.2. Obsługa wyjątków według typu

Zaprezentowane tu klauzule `rescue` obsługują wszystkie wyjątki klasy `StandardError` (lub jej podklas) oraz ignorują wszystkie obiekty klasy `Exception` nienależące do klasy `StandardError`. Aby obsłużyć niestandardowe wyjątki spoza hierarchii klasy `StandardError` lub obsłużyć tylko określone ich typy, w klauzuli `rescue` trzeba umieścić jedną lub więcej klas wyjątków. Poniższa klauzula jest w stanie obsługiwać każdy rodzaj wyjątku:

```
rescue Exception
```

Poniższa klauzula `rescue` obsługuje błędy klasy `ArgumentError` i przypisuje obiekty wyjątków do zmiennej `e`:

```
rescue ArgumentError => e
```

Przypomnijmy, że zdefiniowana wcześniej metoda `factorial` może zgłaszać wyjątki klasy `ArgumentError` i `TypeError`. Poniżej znajduje się klauzula `rescue` obsługująca wyjątki obu tych typów i przypisującą obiekt wyjątku do zmiennej o nazwie `error`:

```
rescue ArgumentError, TypeError => error
```

Tutaj w końcu widać najbardziej ogólną składnię klauzuli `rescue`. Po słowie kluczowym `rescue` może być zero lub więcej oddzielonych przecinkami wyrażeń, z których każde musi zwracać obiekt reprezentujący klasę `Exception` lub jej podkласę. Po tych wyrażeniach można opcjonalnie postawić operator `=>` i nazwę zmiennej.

Teraz wyobraź sobie, że chcesz obsługiwać wyjątki klasy `ArgumentError` i `TypeError`, ale na różne sposoby. Aby wykonać różne bloki kodu w zależności od klasy obiektu wyjątku, można użyć instrukcji `case`. Jednak lepiej będzie wyglądało użycie kilku klauzul `rescue`. Instrukcja `begin` może zawierać zero lub więcej takich klauzul:

```
begin
  x = factorial(1)
rescue ArgumentError => ex
  puts "Spróbuj ponownie z wartością >= 1"
rescue TypeError => ex
  puts "Spróbuj ponownie z liczbą całkowitą"
end
```

Zauważ, że interpreter próbuje dopasować wyjątki do klauzul `rescue` w takiej kolejności, w jakiej zostały zapisane. W związku z tym powinno się ustawić podklasy od najbardziej szczegółowych do najbardziej ogólnych. Aby na przykład obsługiwać wyjątki klasy `EOFError` inaczej niż klasy `IError`, należy najpierw wpisać klauzulę `rescue` dla `EOFError`, ponieważ w przeciwnym wypadku zostaną one obsłużone przez kod klauzuli `IError`. Chcąc utworzyć klauzulę `rescue` obsługującą wszystkie wyjątki nieobsłużone przez wcześniejsze klauzule, na samym końcu należy napisać klauzulę `rescue` `Exception`.

5.6.3.3. Propagacja wyjątków

Po wprowadzeniu klauzul `rescue` można bardziej szczegółowo zająć się propagacją wyjątków. Kiedy zostaje zgłoszony wyjątek, sterowanie zostaje natychmiast przekazane na zewnątrz i do góry, aż do klauzuli `rescue` przeznaczonej do obsługi tego wyjątku. Gdy zostaje wykonana metoda `raise`, interpreter szuka w zawierającym ją bloku związanej z nią klauzuli. Jeśli jej nie znajdzie (lub jeżeli znaleziona klauzula nie jest przeznaczona do obsługi tego rodzaju wyjątków), szuka on w bloku zawierającym ten blok. W sytuacji gdy nie ma odpowiedniej klauzuli `rescue` nigdzie w metodzie, która wywołała metodę `raise`, kończy ona działanie.

Kiedy metoda kończy działanie z powodu wyjątku, nie odbywa się to tak samo jak przy normalnym zwrocie wartości. Taka metoda nie ma wartości zwrotnej, a obiekt wyjątku jest propagowany z miejsca wywołania tej metody. Wyjątek jest przesyłany na zewnątrz przez otaczające go bloki w poszukiwaniu klauzuli `rescue` zadeklarowanej do jego obsługi. Jeśli klauzula taka nie zostanie znaleziona, metoda zwraca wartość do **swojego** wywołującego. Wywołujący kontynuuje w górę stosu. Jeżeli nie zostanie znaleziona żadna procedura obsługi, interpreter drukuje informację o wyjątku i dane ze śledzenia stosu oraz kończy działanie. Przykładem niech będzie poniższy program:

```
def explode          # Niniejsza metoda zgłasza wyjątek klasy RuntimeError w 10% przypadków.
  raise "bam!" if rand(10) == 0
end
def risky
  begin            # Ten blok
    10.times do   # zawiera inny blok,
      explode     # który może zgłosić wyjątek.
    end
    rescue TypeError # Niniejsza klauzula rescue nie może obsłużyć wyjątku RuntimeError,
      puts $!       # a więc zostaje pominięta i wyjątek jest propagowany dalej.
    end
    "hello"        # Normalna wartość zwrotna, jeśli nie wystąpi żaden wyjątek.
  end
  # Tutaj nie ma klauzuli rescue, a więc wyjątek przesyłany jest do wywołującego.
def defuse
  begin            # Niniejszy kod może spowodować wyjątek.
    puts risky      # Próba wywołania oraz wydrukowania wartości zwrotnej.
    rescue RuntimeError => e # Jeżeli został zgłoszony wyjątek,
      puts e.message # w zamian drukujesz komunikat o błędzie.
    end
  end
  defuse
```

Wyjątek jest zgłaszany w metodzie `explode`. Nie posiada ona klauzuli `rescue`, a więc wyjątek zostaje przesłany do wywołującego — metody o nazwie `risky`, która zawiera klauzulę `rescue`, ale obsługującą tylko wyjątki klasy `TypeError`, nie `RuntimeError`. Wyjątek jest propagowany przez bloki leksykalne metody `risky`, aż dochodzi do kodu wywołującego — metody o nazwie `defuse`. Zawiera ona klauzulę `rescue` dla wyjątków klasy `RuntimeError`, a więc sterowanie jest przekazywane do tej klauzuli i propagacja wyjątku zostaje zakończona.

Warto zauważyć, że w powyższym programie została użyta metoda iteracyjna (`Integer.times`) z blokiem. Dla uproszczenia napisaliśmy, że wyjątek jest propagowany na zewnątrz przez ten blok leksykalny. Prawda jest taka, że bloki swoim zachowaniem przy propagacji wyjątków bardziej przypominają wywołania metod. Wyjątek jest propagowany z bloku w górę, do iteratora, który wywołał ten blok. Predefiniowane iteratory pętlowe jak `Integer.times` same nie obsługują wyjątków, dlatego wyjątek jest propagowany w górę stosu wywołań od iteratora `times` do metody `risky`, która go wywołała.

5.6.3.4. Wyjątki występujące podczas obsługi wyjątków

Jeśli w czasie wykonywania kodu w klauzuli `rescue` wystąpi wyjątek, ten, który był pierwotnie obsługiwany, zostaje porzucony i nowy wyjątek jest propagowany z punktu, w którym został zgłoszony. Nowy wyjątek nie może zostać obsłużony przez klauzule `rescue` znajdujące się za tą, w której on wystąpił.

5.6.3.5. Instrukcja retry w klauzuli rescue

Instrukcja `retry` użyta w klauzuli `rescue` zwraca blok kodu, do którego przywiązana jest klauzula `rescue`. Jeśli wyjątek zostanie spowodowany przez jakąś przejściową awarię typu przeciążenie serwera, warto w ramach jego obsługi spróbować wykonać tę operację jeszcze raz. Jednak wiele innych wyjątków jest związanych z błędami programistycznymi (`TypeError`, `ZeroDivisionError`) lub awariami o charakterze stałym (`EOFError` lub `NoMemoryError`). Instrukcja `retry` nie jest właściwą techniką do obsługi tego typu wyjątków.

Poniżej znajduje się przykładowy program, w którym użyto instrukcji `retry` do oczekania na rozwiązywanie problemu z niedziałającą siecią. Próbuje on wczytać treść znajdująca się pod podanym adresem URL, a w razie niepowodzenia próbuje jeszcze raz. Próba ta nie jest powtarzana więcej niż cztery razy, a czas pomiędzy kolejnymi próbami jest zwiększany przy użyciu algorytmu „eksponencjalnego zwiększenia czasu oczekiwania” (ang. *exponential backoff*):

```
require 'open-uri'
tries = 0          # Ile razy próbowało odczytać podany adres URL.
begin             # Początek instrukcji retry.
  tries += 1      # Próba wydrukowania zawartości adresu URL.
  open('http://www.example.com/') {|f| puts f.readlines }
rescue OpenURI::HTTPError => e  # Jeśli wystąpi błąd HTTP,
  puts e.message           # drukuje komunikat o błędzie.
  if (tries < 4)            # Jeżeli nie było jeszcze czterech prób...
    sleep(2**tries)         # czeka 2, 4 i 8 sekund.
    retry                  # A następnie próbuje ponownie!
end
```

5.6.4. Klauzula else

W instrukcji `begin` po klauzuli `rescue` może znajdować się klauzula `else`. Można zgadywać, że jest ona uniwersalnym odpowiednikiem klauzuli `rescue` — obsługuje wszystkie wyjątki, które nie pasują do żadnej z wcześniejszych klauzul `rescue`. Jednak nie do tego została stworzona. Jest ona alternatywą dla klauzuli `rescue`. Używa się jej wówczas, gdy nie jest potrzebna żadna z klauzul `rescue`. To znaczy kod klauzuli `else` jest wykonywany, jeśli kod w ciele instrukcji `begin` zostanie wykonany w całości bez żadnych wyjątków.

Umieszczenie kodu w klauzuli `else` jest jak doczepienie go na końcu klauzuli `begin`. Jedyna różnica jest taka, że żadne wyjątki zgłoszone przez klauzulę `else` nie są obsługiwane przez instrukcję `rescue`.

Klauzula `else` nie jest zbyt często używana w ten sposób, chociaż można za jej pomocą podkreślić różnicę pomiędzy normalnym zakończeniem działania bloku kodu a zakończeniem spowodowanym przez wyjątek.

Warto zauważyc, że użycie klauzuli `else` bez przynajmniej jednej klauzuli `rescue` jest pozbawione sensu. Interpreter zezwala na to, ale zgłasza ostrzeżenie. Za klauzulą `else` nie może być żadnych klauzul `rescue`.

W końcu należy pamiętać, że kod w klauzuli `else` jest wykonywany tylko wówczas, gdy kod w klauzuli `begin` zostaje wykonany do samego końca. Jeśli wystąpi jakiś wyjątek, klauzula `else` oczywiście nie zostanie wykonana. Wykonaniu jej mogą też zapobiec znajdujące się w klauzuli `begin` instrukcje typu `break`, `return` czy `next`.

5.6.5. Klauzula ensure

Instrukcja `begin` może posiadać jedną finalną klauzulę. Opcjonalna klauzula `ensure`, jeżeli zostanie użyta, musi znajdować się za wszystkimi klauzulami `rescue` i `else`. Można jej także używać samodzielnie, bez klauzul `rescue` i `else`.

Kod zawarty w klauzuli `ensure` jest wykonywany zawsze, bez względu na to, co się stanie z kodem za instrukcją `begin`:

- W przypadku gdy kod zostanie wykonany w całości, sterowanie przekazywane jest do klauzuli `else` (jeśli istnieje), a następnie do klauzuli `ensure`.
- Jeżeli w kodzie zostanie wykonana instrukcja `return`, sterowanie pomija klauzulę `else` i przechodzi bezpośrednio do klauzuli `ensure` przed zwróceniem wartości.
- Gdy kod po instrukcji `begin` zgłosi wyjątek, sterowanie przechodzi do odpowiedniej klauzuli `rescue`, a następnie do klauzuli `ensure`.
- Jeśli nie ma ani jednej klauzuli `rescue` lub żadna z istniejących nie może obsłużyć zgłoszonego wyjątku, sterowanie przechodzi bezpośrednio do klauzuli `ensure`. Kod w tej klauzuli jest wykonywany przed propagacją wyjątku do nadrzednego bloku lub w góre stosu wywołań.

Celem stosowania klauzuli `ensure` jest zapewnienie dopilnowania takich niezbędnych czynności, jak zamknięcie plików, rozłączenie z bazą danych czy dokonanie lub anulowanie transakcji. Jest to bardzo przydatna struktura sterująca, której powinno się używać zawsze przy alokacji zasobów (jak uchwyty do plików czy połączenia z bazą danych) w celu zapewnienia prawidłowej ich dealokacji i czyszczenia.

Należy zauważać, że klauzule `ensure` komplikują propagację wyjątków. Pisząc wcześniej o propagacji wyjątków, całkowicie pomineliśmy temat klauzul `ensure`. Propagowany wyjątek nie przeskakuje po prostu w czarodziejski sposób z miejsca, w którym został zgłoszony, do miejsca, gdzie zostanie obsłużony. W takiej sytuacji zachodzi prawdziwy proces propagacji. Interpreter przeszukuje nadrzedne bloki i stos wywołań. W każdej instrukcji `begin` szuka klauzuli `rescue` mogącej obsłużyć zgłoszony wyjątek. Ponadto wyszukuje wszystkie związane z tymi instrukcjami klauzule `ensure` i wykonuje te, przez które przechodzi.

Klauzula `ensure` może anulować propagację wyjątku, inicjując jakiś inny transfer sterowania. Jeżeli klauzula `ensure` zgłosi nowy wyjątek, jest on propagowany zamiast poprzedniego wyjątku. W sytuacji gdy klauzula `ensure` zawiera instrukcję `return`, propagacja wyjątku zostaje zatrzymana i nadrzedna metoda zwraca wartość. Podobny efekt wywołują takie instrukcje sterujące jak `break` i `next` — następuje zatrzymanie propagacji wyjątku i przeniesienie sterowania w odpowiednie miejsce.

Klauzula `ensure` komplikuje także proces zwrotu wartości przez metodę. Mimo iż głównym jej zastosowaniem jest zapewnienie wykonania określonego kodu bez względu na to, czy wystąpi wyjątek, czy nie, klauzulę tę stosuje się także, aby zapewnić wykonanie określonego kodu przed zwróceniem wartości przez wybraną metodę. Jeżeli w ciele instrukcji `begin` znajduje się instrukcja `return`, kod klauzuli `ensure` zostanie wykonany przed zwrotem wartości przez metodę do algorytmu wywołującego. Ponadto jeżeli klauzula `ensure` zawiera własną instrukcję `return`, zmieni wartość zwrotną metody. Na przykład poniższy kod zwraca wartość 2:

```

begin
  return 1      # Przechodzi do klauzuli ensure przed zwrotem wartości do wywołującego.
ensure
  return 2      # Wartość zwrotna zostanie zastąpiona tą nową wartością.
end

```

Należy zauważyc, że klauzula ensure nie zmienia wartości zwrotnej metody, jeśli nie używa jawnie instrukcji return. Na przykład poniższa metoda zwraca wartość 1 zamiast 2:

```

def test
  begin return 1 ensure 2 end
end

```

W przypadku gdy instrukcja begin nie propaguje wyjątku, wartością całej instrukcji jest wartość ostatniego wyrażenia obliczonego w klauzuli begin, rescue lub else. Kod w klauzuli ensure na pewno zwróci wartość, ale nie wpłynie ona na wartość instrukcji begin.

5.6.6. Słowo kluczowe rescue w definicjach metod, klas i modułów

Opisując temat obsługi wyjątków, przedstawiliśmy słowa kluczowe rescue, else i ensure jako klauzule instrukcji begin. Można ich jednak używać także jako klauzuli instrukcji def (definiującej metody), class (definiującej klasy) oraz module (definiującej moduły). Definowanie metod zostało opisane w rozdziale 6., a klas i modułów w rozdziale 7.

Poniżej znajduje się schematyczna definicja metody z klauzulami rescue, else i ensure:

```

def method_name(x)
  # Tutaj znajduje się ciało metody.
  # Zazwyczaj kod ten jest wykonywany do samego końca, bez żadnych wyjątków
  # i normalnie zwraca wartość do swojego wywołującego.

rescue
  # W tym miejscu znajduje się kod obsługi błędów.
  # Jeśli w ciele metody zostanie zgłoszony wyjątek lub jeżeli
  # jedna z wywołanych w tym ciele metod zgłosi wyjątek, sterowanie
  # zostaje przekazane do tego bloku.

else
  # Jeśli w ciele metody nie wystąpią żadne wyjątki,
  # zostanie wykonany kod tej klauzuli.

ensure
  # Niniejszy kod zostanie wykonany bez względu na to, co się stanie w
  # ciele metody. Jest on wykonywany, gdy metoda zostanie w całości wykonana, jeżeli
  # wyrzuci wyjątek lub wykona instrukcję return.

end

```

5.6.7. Słowo kluczowe rescue w roli modyfikatora instrukcji

Słowo kluczowe rescue poza klauzulą może też być używane jako modyfikator instrukcji. Po każdej instrukcji może znajdować się słowo kluczowe rescue i inne wyrażenie. W przypadku gdy pierwsza instrukcja zgłosi wyjątek, w zamian wykonywana jest druga. Na przykład:

```

# Oblicza silnię x lub używa wartości 0, jeśli metoda zgłosi wyjątek.
y = factorial(x) rescue 0

```

Kod ten jest równoznaczny z poniższym:

```

y = begin
  factorial(x)
rescue
  0
end

```

Zaletą składni modyfikatora instrukcji jest to, że nie są wymagane słowa kluczowe begin i end. Słowo kluczowe rescue użyte w ten sposób musi być samo, bez nazw klas wyjątków i zmiennych. Modyfikator rescue obsługuje wszystkie wyjątki klasy StandardError i żadnych innych. W przeciwnieństwie do modyfikatorów if i while modyfikator rescue ma wyższy priorytet (zobacz tabelę 4.2 w poprzednim rozdziale) niż operatory przypisania. To znaczy że ma on zastosowanie tylko do prawej strony przypisania (jak w powyższym przykładzie), a nie do wyrażenia przypisania jako całości.

5.7. Instrukcje BEGIN i END

BEGIN i END to słowa zarezerwowane służące do deklarowania kodu, który ma być wykonany na samym początku i końcu programu (należy zauważać, że słowa BEGIN i END pisane wielkimi literami oznaczają coś całkiem innego niż słowa begin i end pisane małymi literami). Jeśli w programie znajduje się więcej niż jedna instrukcja BEGIN, interpreter wykonuje je w takiej kolejności, w jakiej zostały napisane. Gdy w programie jest więcej niż jedna instrukcja END, interpreter wykonuje je w odwrotnej kolejności, niż zostały wpisane — to znaczy pierwsza z nich jest wykonywana na końcu. Instrukcje te nie są powszechnie używane w Ruby. Zostały one odziedziczone po Perlu, który z kolei odziedziczył je po języku przetwarzania tekstu awk.

Po słowach kluczowych BEGIN i END musi znajdować się otwierająca klamra, dowolna ilość kodu źródłowego oraz klamra zamkająca. Klamry są wymagane — nie można ich zastąpić słowami kluczowymi do i end. Na przykład:

```
BEGIN {  
    # Globalny kod inicjujący.  
}  
END {  
    # Globalny kod kończący.  
}
```

Instrukcje BEGIN i END różnią się od siebie pewnymi szczegółami. Instrukcje BEGIN są wykonywane na samym początku, nawet przed kodem znajdującym się przed nimi. Oznacza to, że tworzą one lokalny zakres działania dla zmiennych, który jest całkowicie oddzielony od otaczającego je kodu. Rozsądne jest umieszczenie instrukcji BEGIN tylko na samym początku kodu. Instrukcje te znajdujące się wewnętrz instrukcji warunkowych i pętli są wykonywane na początku bez względu na otaczający je kod. Spójrz na poniższy kod:

```
if (false)  
BEGIN {  
    puts "pętla";  
    a = 4;  
}  
else  
    BEGIN { puts "if" }  
end  
10.times {BEGIN { puts "else" }} # To zostanie wydrukowane tylko jeden raz.
```

Kod związany z wszystkimi trzema instrukcjami BEGIN zostanie wykonany tylko jeden raz, bez względu na otoczenie, w którym się znajduje. Zmienne zdefiniowane w blokach BEGIN nie są widoczne na zewnątrz tych bloków i żadne zmienne poza tym blokiem nie są podczas jego wykonywania definiowane.

Instrukcje END są inne. Są wykonywane w toku normalnego wykonywania programu, a więc współdzielą zmienne lokalne z otaczającym je kodem. Jeśli instrukcja END znajduje się w nigdy niewykonywanej instrukcji warunkowej, kod z nią związany nie jest rejestrowany do wykonania

przed zamknięciem programu. W przypadku gdy instrukcja END znajduje się w pętli i jest wykonywana więcej niż jeden raz, kod z nią związany jest mimo to rejestrowany tylko jeden raz:

```
a = 4;
if (true)
  END {
    puts "petla";
    puts a
  }
else
  END { puts "if" }           # Ten kod nie jest wykonywany.
end
10.times {END { puts "else" }} # To jest wykonywane tylko jeden raz.
```

Ta instrukcja END jest wykonywana.
Ten kod zostaje zarejestrowany.
Ta zmienna jest widoczna; drukuje „4”.

Alternatywą dla instrukcji END jest metoda at_exit z modulu Kernel. Rejestruje blok kodu do wykonania bezpośrednio przed zakończeniem pracy interpretera. Podobnie jak w przypadku bloków END, kod związany z pierwszą metodą at_exit zostanie wykonany na końcu. Jeśli metoda at_exit zostanie wywołana kilka razy w pętli, związany z nią blok kodu zostanie również wywołany kilka razy przed zakończeniem działania interpretera.

5.8. Wątki, włókna i kontynuacje

Niniejszy podrozdział wprowadza wątki pozwalające na współbieżne wykonywanie działań oraz dwie bardziej egzotyczne konstrukcje sterujące o nazwie włókien i kontynuacji.

5.8.1. Współbieżność wątkowa

Wątek wykonawczy to szereg instrukcji, które działają (lub wydają się działać) równolegle z głównymi instrukcjami wykonywanymi przez interpreter. Wątki są reprezentowane przez obiekty klasy Thread, ale można je także traktować jako instrukcje sterujące współbieżności. Szczegółowy opis technik programowania współbieżnego znajduje się w podrozdziale 9.9. Niniejszy podrozdział zawiera tylko krótki przegląd tych technik i informacje na temat tworzenia wątków.

Dzięki własnościom bloków tworzenie wątków jest w języku Ruby bardzo proste. Wystarczy wywołać metodę Thread.new i związać z nią dowolny blok kodu. W wyniku tego zostanie utworzony nowy wątek i rozpoczęcie się wykonywanie kodu w jego bloku. Tymczasem oryginalny wątek zwróci wartość z wywołania Thread.new i będzie kontynuował wykonywanie kolejnych instrukcji. Nowo utworzony wątek zakończy działanie w chwili zakończenia bloku, którego wartość zwrotna będzie dostępna poprzez metodę value obiektu klasy Thread (jeśli metoda ta zostanie wywołana przed zakończeniem działania wątku, mechanizm wywołujący zostanie zablokowany do czasu, aż wątek ten zwróci wartość).

Poniższy kod demonstruje równoczesne wczytywanie kilku plików przy użyciu wątków:

```
# Niniejsza metoda pobiera tablicę nazw plików.
# Zwraca tablicę łańcuchów reprezentujących treść podanych plików.
# Metoda ta tworzy po jednym wątku dla każdego przekazanego do niej pliku.
def readfiles(filenames)
  # Utworzenie tablicy wątków z tablicy nazw plików.
  # Każdy wątek zaczyna odczytywać plik.
  threads = filenames.map do |f|
    Thread.new { File.read(f) }
  end
  # Teraz zostanie utworzona tablica dla zawartości plików poprzez wywołanie metody
```

```

# value każdego z wątków. Metoda ta zostaje w razie potrzeby zablokowana,
# aż wątek zakończy działanie i wróci wartość.
threads.map { |t| t.value }
end

```

Więcej informacji na temat wątków i współbieżności w języku Ruby znajduje się w podrozdziale 9.9.

5.8.2. Włókna i współprogramy

W Ruby 1.9 wprowadzono struktury sterujące o nazwie **włókien**, które są reprezentowane przez obiekty klasy `Fiber`. Nazwa „włókno” w innych językach oznacza pewien rodzaj lekkiego wątku, ale włókna w języku Ruby lepiej opisać jako **współprogramy** lub bardziej precyzyjnie **semiwspółprogramy**. Współprogramy są najczęściej wykorzystywane do implementacji generatorów, czyli obiektów obliczających częściowy wynik, zwracających ten wynik do struktury, która je wywołała, oraz zapisujących stan tych obliczeń, aby mechanizm wywołujący mógł je wznowić w celu uzyskania kolejnego wyniku. Klasa `Fiber` pozwala na automatyczną konwersję wewnętrznych iteratorów, podobnie jak metoda `each`, na enumeratory lub iteratory zewnętrzne.

Włókna są zaawansowaną i niezbyt często używaną strukturą sterującą. Większość programistów Ruby nigdy nie będzie potrzebowała bezpośrednio używać klasy `Fiber`. Dla osób, które nigdy wcześniej nie używały w swoich programach współprogramów i generatorów, zrozumienie zasady działania włókien może okazać się za pierwszym razem trudne. W takim przypadku należy uważnie przeanalizować przedstawione przykłady i wypróbować kilka własnych programów.

Włókno posiada własny blok kodu, podobnie jak wątek. Nowe włókno tworzy się za pomocą metody `Fiber.new`. Kod wykonywany przez włókno należy umieścić w związanym z włóknem bloku. Ciało włókna, w przeciwieństwie do wątku, nie jest wykonywane natychmiast. Aby uruchomić włókno, należy wywołać metodę `resume` obiektu klasy `Fiber`, który je reprezentuje. Przy pierwszym wywołaniu metody `resume` na rzecz włókna sterowanie jest przekazywane na początek jego ciała. Tak uruchomione włókno działa aż do napotkania końca swojego ciała lub wykonania metody klasowej `Fiber.yield` przekazującej sterowanie z powrotem do algorytmu wywołującego i zmuszającej metodę `resume` do zwrócenia wartości. Dodatkowo metoda `Fiber.yield` zapisuje stan włókna, dzięki czemu kolejne wywołanie metody `resume` powoduje uruchomienie włókna od momentu, w którym zostało wcześniej zatrzymane. Oto prosty przykład:

```

f = Fiber.new {
  puts "Włókno wita się"
  Fiber.yield
  puts "Włókno żegna się"
}

puts "Algorytm wywołujący wita się"
f.resume
puts "Algorytm wywołujący żegna się"
f.resume

```

Wiersz 1.: utworzenie nowego włókna.
Wiersz 2.:
Wiersz 3.: przejście do wiersza 9.
Wiersz 4.:
Wiersz 5.: przejście do wiersza 11.
Wiersz 6.:
Wiersz 7.:
Wiersz 8.: przejście do wiersza 2.
Wiersz 9.:
Wiersz 10.: przejście do wiersza 4.
Wiersz 11.:

Ciało włókna nie jest uruchamiane w miejscu jego utworzenia. Dlatego powyższy kod tworzy włókno, ale nie daje żadnego wyniku aż do napotkania wiersza 7. Następnie wywołania metod `resume` i `Fiber.yield` przekazują sterowanie w tą i z powrotem, dzięki czemu komunikaty z włókna i algorytmu wywołującego przeplatają się. Niniejszy kod zwraca następujący wynik:

```
Algorytm wywołujący wita się
Włókno wita się
Algorytm wywołujący żegna się
Włókno żegna się
```

Warto zauważyc, że działanie metody `Fiber.yield` jest całkiem inne niż działanie instrukcji `yield`. Metoda `Fiber.yield` przekazuje sterowanie z bieżącego wątku z powrotem do algorytmu, który je wywołał. Natomiast instrukcja `yield` przekazuje sterowanie z iteratora do związanego z nim bloku.

5.8.2.1. Argumenty wątków i wartości zwrotne

Wątki i algorytmy je wywołujące mogą wymieniać się danymi poprzez argumenty i wartości zwrotne metod `resume` i `yield`. Argumenty pierwszego wywołania metody `resume` są przekazywane do bloku związanego z wątkiem — stają się wartościami parametrów jego bloku. W kolejnych wywołaniach argumenty metody `resume` stają się wartością zwrotną metody `Fiber.yield`. Natomiast argumenty metody `Fiber.yield` stają się wartością zwrotną metody `resume`. Kiedy zakończy się działanie bloku, wartość ostatniego wyrażenia również staje się wartością zwrotną metody `resume`. Demonstruje to poniższy program:

```
f = Fiber.new do |message|
  puts "Algorytm wywołujący mówi: #{message}"
  message2 = Fiber.yield("Witaj")      # Łącuch "Witaj" zwrócony przez pierwsze wywołanie metody resume.
  puts "Algorytm wywołujący mówi: #{message2}"
  "W porządku"                      # Łącuch "W porządku" zwrócony przez drugie wywołanie metody resume.
end
response = f.resume("Witaj")          # Łącuch "Witaj" przekazany do bloku.
puts "Włókno mówi: #{response}"
response2 = f.resume("Jak leci?")     # Łącuch "Jak leci?" zwrócony przez wywołanie metody Fiber.yield.
puts "Włókno mówi: #{response2}"
```

Algorytm wywołujący przekazuje do wątku dwa komunikaty, a te z kolei zwracają do niego dwie odpowiedzi. Program drukuje następujący dialog:

```
Algorytm wywołujący mówi: Witaj
Włókno mówi: Witaj
Algorytm wywołujący mówi: Jak leci?
Włókno mówi: W porządku
```

W kodzie algorytmu wywołującego komunikaty są argumentami metody `resume`, a odpowiedzi wartością zwrotną tej metody. W ciele wątku wszystkie komunikaty poza pierwszym są odbierane jako wartość zwrotną metody `Fiber.yield`, a wszystkie odpowiedzi z wyjątkiem ostatniej są przekazywane jako argumenty do metody `Fiber.yield`. Pierwszy komunikat zostaje odebrany za pośrednictwem parametrów bloku, a ostatnia odpowiedź jest wartością zwrotną samego bloku.

5.8.2.2. Implementacja generatorów przy użyciu wątków

Przedstawione do tej pory przykłady użycia wątków nie należą do najbardziej realistycznych. Tym razem zaprezentujemy bardziej typowy sposób ich wykorzystania. Najpierw napisany zostanie generator liczb Fibonacciego — obiekt klasy `Fiber` zwracający przy każdym wywołaniu metody `resume` kolejne elementy ciągu Fibonacciego:

```
# Zwraca obiekt klasy Fiber obliczający liczby Fibonacciego.
def fibonacci_generator(x0,y0)    # Oparcie ciągu na x0,y0.
  Fiber.new do
    x, y = x0, y0                  # Inicjacja x i y.
    loop do                         # To wólkno działa wiecznie.
      ...
```

```

    Fiber.yield y           # Zwrót kolejnej liczby ciągu.
    x,y = y,x+y            # Aktualizacja x i y.
  end
end
g = fibonacci_generator(0,1)      # Utworzenie generatora.
10.times { print g.resume, " " } # Użycie generatora.

```

Powyższy program drukuje dziesięć pierwszych liczb Fibonacciego:

```
1 1 2 3 5 8 13 21 34 55
```

Ponieważ klasa `Fiber` jest zawiązała strukturą sterującą, dobrym pomysłem może być ukrycie jej API podczas pisania generatora. Poniżej znajduje się inna wersja generatora liczb Fibonacciego. Definiuje on własną klasę oraz implementuje te same API metod `next` i `rewind` co enumeratory:

```

class FibonacciGenerator
  def initialize
    @x,@y = 0,1
    @fiber = Fiber.new do
      loop do
        @x,@y = @y, @x+@y
        Fiber.yield @x
      end
    end
  end
  def next          # Zwrót kolejnej liczby Fibonacciego.
    @fiber.resume
  end
  def rewind        # Ponowne uruchomienie ciągu.
    @x,@y = 0,1
  end
end
g = FibonacciGenerator.new      # Utworzenie generatora.
10.times { print g.next, " " } # Wydruk pierwszych dziesięciu liczb.
g.rewind; puts                 # Rozpoczęcie od nowa w nowym wierszu.
10.times { print g.next, " " } # Ponowny wydruk dziesięciu pierwszych liczb.

```

Warto zauważać, że można sprawić, aby powyższa klasa `FibonacciGenerator` była przeliczalna, dodając do niej moduł `Enumerable` i dodając poniższą metodę `each` (która po raz pierwszy została użyta w podrozdziale 5.3.5):

```

def each
  loop { yield self.next }
end

```

Z drugiej strony wyobraź sobie, że masz obiekt `Enumerable` i chcesz z niego zrobić generator. Możesz w takiej sytuacji użyć poniższej klasy:

```

class Generator
  def initialize(enumerable)
    @enumerable = enumerable # Zapamiętanie obiektu przeliczalnego.
    create_fiber            # Utworzenie wątku służącego do jego przeliczania.
  end
  def next                # Zwrót kolejnego elementu
    @fiber.resume          # przez uruchomienie wątku.
  end
  def rewind              # Ponowne uruchomienie enumeracji
    create_fiber          # poprzez utworzenie nowego wątku.
  end
  private
  def create_fiber         # Utworzenie wątku odpowiedzialnego za enumerację.
    @fiber = Fiber.new do
      @enumerable.each do |x| # Użycie metody each.
        Fiber.yield(x)       # Wstrzymanie enumeracji w celu zwrotu wartości.
      end
    end

```

```

        raise StopIteration      # Zgłoszenie wyjątku, kiedy nie ma już wartości.
    end
end
g = Generator.new(1..10)  # Utworzenie generatora.
loop { print g.next }    # Użycie go jak enumeratatora.
g.rewind                 # Rozpoczęcie od początku.
g = (1..10).to_enum      # Metoda to_enum robi to samo.
loop { print g.next }

```

Mimo iż przeanalizowanie implementacji tej klasy jest korzystne, sama klasa nie udostępnia większej funkcjonalności niż metoda `to_enum`.

5.8.2.3. Zaawansowane właściwości wątków

Dostępny w bibliotece standardowej moduł `fiber` nadaje wątkom dodatkowe, bardziej zaawansowane funkcje. Aby móc z nich korzystać, należy w źródle programu wpisać poniższy wiersz kodu:

```
require 'fiber'
```

Tych dodatkowych funkcji należy jednak unikać, ponieważ:

- Nie są one obsługiwane przez wszystkie implementacje. Na przykład JRuby nie obsługuje ich w aktualnych maszynach wirtualnych Javy.
- Są tak potężne, że nieprawidłowe ich użycie może spowodować awarię maszyny wirtualnej Ruby.

Rdzenne właściwości klasy `Fiber` implementują semiwspółprogramy (ang. *semicoroutines*). Nie są to prawdziwe współprogramy, ponieważ pomiędzy algorytmem wywołującym a wątkiem występuje podstawowa asymetria — algorytm wywołujący używa metody `resume`, a wątko `yield`. Jeśli jednak zostanie dołączona biblioteka `fiber`, klasa `Fiber` zyska metodę `transfer`, która pozwala każdemu wątkowi przenieść sterowanie do dowolnego innego wątkna. Poniżej znajduje się przykład programu, w którym dwa wątki przekazują sterowanie (i wartości) w tą i z powrotem za pomocą metody `transfer`:

```

require 'fiber'
f = g = nil
f = Fiber.new { |x|
  puts "f1: #{x}"          # 1:
  x = g.transfer(x+1)      # 2: drukuje "f1: 1".
  puts "f2: #{x}"          # 3: przekazuje 2 do wiersza 8.
  x = g.transfer(x+1)      # 4: drukuje "f2: 3".
  puts "f3: #{x}"          # 5: zwraca 4 do wiersza 10.
  puts "f4: #{x}"          # 6: drukuje "f3: 5".
  x + 1                   # 7: zwraca 6 do wiersza 13.
}
g = Fiber.new { |x|
  puts "g1: #{x}"          # 8:
  x = f.transfer(x+1)      # 9: drukuje "g1: 2".
  puts "g2: #{x}"          # 10: zwraca 3 do wiersza 3.
  x = f.transfer(x+1)      # 11: drukuje "g2: 4".
  puts "g3: #{x}"          # 12: zwraca 5 do wiersza 5.
}
puts f.transfer(1)          # 13: przekazuje 1 do wiersza 1.

```

Niniejszy kod daje poniższy wynik:

```
f1: 1
g1: 2
f2: 3
g2: 4
f3: 5
6
```

Większość osób nigdy nie będzie potrzebować metody `transfer`, ale jej przedstawienie po-maga wyjaśniać znaczenie nazwy `fiber`. Włókna można traktować jako niezależne ścieżki wykonawcze w obrębie jednego wątku wykonawczego. Jednak w przeciwnieństwie do wątków włókna nie posiadają algorytmu planującego przekazującego sterowanie pomiędzy nimi. Uruchamianie włókien musi być planowane jawnie za pomocą metody `transfer`.

Poza metodą `transfer` biblioteka `fiber` definiuje także metodę obiektową `alive?` sprawdzającą, czy ciało włókna jest jeszcze wykonywane, oraz metodę klasową `current` zwracającą obiekt klasy `Fiber`, który ma aktualnie kontrolę.

5.8.3. Kontynuacja

Kontynuacja (ang. *continuation*) to kolejna skomplikowana i rzadko używana struktura sterująca, której większość programistów nigdy nie używa. Kontynuacja ma postać metody o nazwie `callcc` z modułu `Kernel` i obiektu klasy `Continuation`. Kontynuacje wchodzą w skład rdzenia Ruby 1.8, a w Ruby 1.9 zostały zastąpione włóknami i przeniesione do biblioteki standardowej. Aby użyć ich w Ruby 1.9, trzeba dodać do programu poniższy wiersz kodu:

```
require 'continuation'
```

Trudności implementacyjne kontynuacji sprawiają, że inne implementacje Ruby (jak oparta na Javie implementacja Ruby o nazwie JRuby) nie obsługują ich. Ponieważ kontynuacje nie są już dobrze obsługiwane, należy je traktować jako ciekawostkę i nigdy nie stosować ich w nowych programach. Programy napisane w Ruby 1.8 używające kontynuacji często można dostosować do Ruby 1.9 przy użyciu włókien.

Metoda `callcc` z modułu `Kernel` wykonuje swój blok kodu i przekazuje nowo utworzony obiekt klasy `Continuation` jako jedyny argument. Obiekty klasy `Continuation` udostępniają metodę `call` zmuszającą metodę `callcc` do zwrotu wartości do algorytmu, który ją wywołał. Wartość przekazana do metody `call` staje się wartością zwrotną metody `callcc`. W tym sensie `callcc` działa jak instrukcja `catch`, a `call` obiektu klasy `Continuation` jak instrukcja `throw`.

Kontynuacje są jednak inne, ponieważ obiekt klasy `Continuation` można zapisać w zmiennej poza blokiem metody `callcc`. Metodę `call` tego obiektu można wywoływać wielokrotnie, co powoduje przeskok sterowania do pierwszej instrukcji znajdującej się za wywołaniem metody `callcc`.

Poniższy program demonstruje sposób użycia kontynuacji do zdefiniowania metody działającej tak samo jak instrukcja `goto` w języku programowania BASIC:

```
# Globalna tablica asocjacyjna wiążąca numery wierszy (lub symbole) z kontynuacjami.
$lines = {}
# Utworzenie kontynuacji i powiązanie jej z określonym numerem wiersza.
def line(symbol)
  callcc {|c| $lines[symbol] = c }
end
# Odszukanie kontynuacji związanej z tym numerem i przeskoczenie do niej.
def goto(symbol)
  $lines[symbol].call
end
# Teraz możesz udawać, że programujesz w języku BASIC.
i = 0
line 10          # Deklaracja tego miejsca jako wiersza numer 10.
puts i += 1
goto 10 if i < 5 # Przeskok z powrotem do wiersza numer 10, jeśli warunek został spełniony.
line 20          # Deklaracja miejsca jako wiersz numer 20.
puts i -= 1
goto 20 if i > 0
```

Metody, obiekty klasy Proc, lambdy i domknięcia



Metoda to blok parametryzowanego kodu związanego z co najmniej jednym obiektem. **Wywołanie** metody składa się z nazwy wywoływanej metody, obiektu, na rzecz którego ma ona być wywołana (czasami nazywanego **adresatem** lub **odbiorcą**), oraz zera lub więcej wartości argumentów przypisywanych do parametrów. Wartość ostatniego wyrażenia w metodzie jest wartością wyrażenia wywołania metody.

W wielu językach programowania rozróżnia się funkcje, z którymi nie jest związany żaden obiekt, i metody wywoływane na rzecz obiektów odbiorców. Ponieważ Ruby jest językiem czysto obiektowym, wszystkie jego metody są prawdziwymi metodami i każda z nich jest związana przynajmniej z jednym obiektem. Ponieważ nie zostały jeszcze wprowadzone klasy, przykładowe metody prezentowane w niniejszym rozdziale wyglądają jak funkcje globalne, z którymi nie jest związany żaden obiekt. W rzeczywistości Ruby niejawnie definiuje je i wywołuje jako prywatne metody klasy `Object`.

Metody są podstawowym komponentem składni języka Ruby, ale nie są wartościami, na których programy mogą wykonywać działania. To znaczy metody nie są obiektami, jak łańcuchy, liczby i tablice. Jest jednak możliwe utworzenie obiektu klasy `Method` reprezentującego wybraną metodę oraz wywoływanie metod pośrednio poprzez obiekty tej klasy.

Metody nie są w języku Ruby jedynym rodzajem parametryzowanego wykonywalnego kodu. Opisane w podrozdziale 5.4 bloki również są wykonywalne i mogą mieć parametry. Bloki w przeciwnieństwie do metod nie mają nazw i mogą być wywoływane tylko pośrednio poprzez metody iteracyjne.

Bloki, podobnie jak metody, nie są obiektami, którymi można manipulować. Jest jednak możliwe utworzenie obiektu reprezentującego blok i możliwość ta jest nawet dość często wykorzystywana w programach Ruby. Obiekty klasy `Proc` reprezentują bloki. Podobnie jak obiekty klasy `Method`, kod bloku można wykonać poprzez reprezentujący go obiekt klasy `Proc`. Istnieją dwa rodzaje obiektów klasy `Proc` — obiekty `proc` i `lambda`, które różnią się nieco sposobem działania. Zarówno `proc`, jak i `lambda` są funkcjami, a nie metodami wywoływanymi na rzecz obiektów. Ważną ich cechą jest to, że są **domknięciami** (ang. *closure*) — zachowują dostęp do zmiennych lokalnych, które były w ich zasięgu, kiedy je definiowano, nawet jeśli obiekty te są wywoływane w innym zakresie.

Składnia metod w języku Ruby jest bogata i skomplikowana, dlatego poświęcono im całe cztery pierwsze podrozdziały niniejszego rozdziału. W pierwszym z nich opisane jest definiowanie prostych metod. W kolejnych zostały zawarte bardziej zaawansowane tematy, jak reguły nazewnicze metod, zasady dotyczące stosowania nawiasów i parametry. Warto zauważyć, że wywołanie metody jest rodzajem wyrażenia, o których była mowa w podrozdziale 4.4. Więcej informacji na temat wywoływania metod znajduje się we wszystkich czterech pierwszych podrozdziałach niniejszego rozdziału.

Po metodach przychodzi kolej na obiekty `proc` i `lambda`. Opisane zostały sposoby ich tworzenia i wywoływania, wyjaśniono również subtelne różnice pomiędzy nimi. Osobny podrozdział został poświęcony używaniu obiektów `proc` i `lambda` jako domknięć. Dalej znajduje się podrozdział poświęcony obiektom klasy `Method`, które zachowują się podobnie jak `lambda`. Rozdział kończy się zaawansowanym opisem programowania funkcyjnego w języku Ruby.

6.1. Definiowanie prostych metod

Do tej pory przedstawionych zostało wiele przykładów wywołań metod, a składnia służąca do ich wywoływania opisana została szczegółowo w podrozdziale 4.4. Teraz zajmiemy się składnią definicji metod — w tym podrozdziale tylko podstawami. Kolejne trzy podrozdziały opisują metody: reguły ich nazywania, zasady używania nawiasów w ich definicjach oraz ich argumenty. Pozostałe podrozdziały dotyczą bardziej zaawansowanych zagadnień i odnoszą się zarówno do definicji, jak i wywołań metod.

Do definiowania metod służy słowo kluczowe `def`. Po nim musi znajdować się nazwa metody oraz otoczona nawiasami opcjonalna lista nazw parametrów. Kod stanowiący ciało metody znajduje się za listą parametrów, a koniec metody wyznacza słowo kluczowe `end`. Nazw parametrów można używać w ciele metody jako zmiennej. Wartości tych parametrów są pobierane z argumentów podawanych przy wywoływaniu metody. Poniżej znajduje się przykładowa metoda:

```
# Definicja metody o nazwie factorial z jednym parametrem o nazwie n.
def factorial(n)
    if n < 1                      # Sprawdzenie, czy wartość argumentu jest poprawna.
        raise "argument musi być > 0"
    elsif n == 1                    # Jeżeli argument ma wartość 1,
        1                          # wartością wywołania metody jest 1.
    else                           # W przeciwnym przypadku silnia n wynosi n razy
        n * factorial(n-1)         # factorial n-1.
    end
end
```

Powyższy kod definiuje metodę o nazwie `factorial` z jednym parametrem o nazwie `n`. Identyfikator `n` jest używany w ciele metody jako nazwa zmiennej. Metoda `factorial` jest metodą rekursywną, a więc w jej ciele znajduje się wywołanie jej samej. Wywołanie składa się z nazwy metody i wartości argumentu zapisanej w nawiasach za tą nazwą.

6.1.1. Wartość zwrotna metody

Metoda może zakończyć działanie w sposób normalny i nienormalny. Ta druga sytuacja ma miejsce wówczas, gdy metoda zgłasza wyjątek. Przedstawiona wcześniej metoda `factorial` kończy działanie w nienormalny sposób, jeśli przekazany do niej argument ma wartość mniejszą niż 1. W sytuacji gdy metoda zakończy działanie w normalny sposób, wartością wyrażenia wywołania metody jest wartość ostatniego wyrażenia obliczonego w ciele tej metody. W metodzie `factorial` ostatnim wyrażeniem może być 1 albo `n*factorial(n-1)`.

Słowo kluczowe `return` wymusza zwrot wartości przez metodę przed jej zakończeniem. Jeżeli po słowie tym znajduje się wyrażenie, zwracana jest wartość tego wyrażenia. Jeśli nie ma po nim żadnego wyrażenia, zwarcana jest wartość `nil`. W poniższej wersji metody `factorial` słowo kluczowe `return` nie może zostać usunięte:

```
def factorial(n)
    raise "zły argument" if n < 1
    return 1 if n == 1
    n * factorial(n-1)
end
```

Słowo kluczowe `return` można także wstawić w ostatnim wierszu ciała metody, aby podkreślić, że dane wyrażenie wyznacza wartość zwrotną metody. Zazwyczaj jednak słowo to nie jest stosowane, jeżeli nie jest to konieczne.

Metody w języku Ruby mogą zwracać więcej niż jedną wartość. Aby tak się stało, należy w instrukcji `return` wpisać dowolną liczbę wartości, które mają zostać zwrócone:

```
# Konwersja punktu kartezjańskiego (x,y) na współrzędne biegunkowe.  
def polar(x,y)  
    return Math.hypot(y,x), Math.atan2(y,x)  
end
```

Jeśli wartości zwrotnych jest więcej niż jedna, są one zapisywane w tablicy, która staje się pojęciem wartością zwrotną metody. Zamiast używać instrukcji `return`, można własnoręcznie utworzyć tablicę wartości:

```
# Konwersja współrzędnych biegunkowych na kartezjańskie.  
def cartesian(magnitude, angle)  
    [magnitude*Math.cos(angle), magnitude*Math.sin(angle)]  
end
```

Metody w takiej postaci są zazwyczaj przeznaczone do użytku w przypisaniach równoległych (zobacz podrozdział 4.5.5), w których każda wartość zwrotna jest przypisywana osobnej zmiennej:

```
distance, theta = polar(x,y)  
x, y = cartesian(distance,theta)
```

6.1.2. Metody a obsługa wyjątków

Instrukcja `def` definiująca metodę może, podobnie jak instrukcja `begin`, zawierać kod obsługujący wyjątki w formie klauzul `rescue`, `else` i `ensure`. Klauzule te wstawia się za ciałem metody, ale przed słowem kluczowym `end` kończącym jej definicję. W krótkich metodach bardzo zgrabnym rozwiązaniem jest związanie klauzul `rescue` z instrukcją `def`. Dzięki temu nie trzeba stosować instrukcji `begin` i związanej z nią wcinania wierszy. Więcej szczegółów na ten temat znajduje się w podrozdziale 5.6.6.

6.1.3. Wywoływanie metod na rzecz obiektów

Metody są zawsze wywoływane na rzecz obiektów (obiekt jest czasami nazywany adresatem przez analogię do paradygmatu programowania obiektowego, w którym metody są nazywane komunikatami wysyłanymi do obiektów adresatów). Słowo kluczowe `self` znajdujące się w ciele metody odnosi się do obiektu, na rzecz którego metoda ta została wywołana. Jeżeli podczas wywoływania metody nie zostanie wyznaczony żaden obiekt, zostanie ona wywołana na rzecz obiektu `self`.

O definiowaniu metod dla klas obiektów piszemy w rozdziale 7. Należy jednak zauważyc, że do tej pory zostało przedstawionych już kilka przykładów wywołań metod na rzecz obiektów, jak poniżej:

```
first = text.index(pattern)
```

Podobnie jak w większości obiektowych języków programowania, w Ruby obiekt, na rzecz którego wywoływana jest metoda, oddziela się od niej kropką. Powyższa procedura przesyła wartość zmiennej `pattern` do metody o nazwie `index` obiektu zapisanego w zmiennej `text`, a następnie zapisuje zwróconą wartość w zmiennej `first`.

6.1.4. Definiowanie metod singletonowych

Wszystkie zdefiniowane do tej pory metody są globalne. Jeśli instrukcja `def`, taka jak te zaprezentowane wcześniej, zostanie wstawiona do instrukcji `class`, powstanie metoda egzemplarza (obiektu) tej klasy. Metoda taka jest dostępna we wszystkich obiektach będących egzemplarzami tej klasy (klasy i metody obiektowe zostały opisane w rozdziale 7.).

Możliwe jest również zdefiniowanie za pomocą słowa kluczowego `def` metody tylko dla jednego wybranego obiektu. W tym celu należy po słowie kluczowym `def` wstawić wyrażenie, którego wartością jest obiekt. Po tym wyrażeniu powinna znajdować się kropka i nazwa metody, która ma zostać zdefiniowana. Powstała w ten sposób metoda jest nazywana **metodą singletonową**, ponieważ jest dostępna tylko dla jednego obiektu:

```
o = "message"      # Łanicz jest obiektem.  
def o.printme     # Definicja metody singletonowej dla tego obiektu.  
  puts self  
end  
o.printme        # Wywolanie metody singletonowej.
```

Metody klasowe (opisane w rozdziale 7.), jak `Math.sin` czy `File.delete`, są metodami singletonowymi. `Math` to stała odwołująca się do obiektu `Module`, a `File` to stała odwołująca się do obiektu `Class`. Obiekty te udostępniają odpowiednio metody `sin` i `delete`.

Wartości klas `Fixnum` i `Symbol` są w różnych implementacjach języka Ruby traktowane jako bezpośrednie wartości, a nie prawdziwe referencje do obiektów (zobacz podrozdział 3.8.1.1). Dlatego metody singletonowe nie mogą być definiowane na obiektach klas `Fixnum` i `Symbol`. Aby zachować konsekwencję, metod singletonowych nie można definiować także dla obiektów klas `Numeric`.

6.1.5. Usuwanie definicji metod

Metody są definiowane za pomocą słowa kluczowego `def`, a do usuwania ich definicji służy instrukcja `undef`:

```
def sum(x,y); x+y; end      # Definicja metody.  
puts sum(1,2)                 # Użycie metody.  
undef sum                      # Oddefiniowanie metody.
```

W powyższym kodzie instrukcja `def` definiuje globalną metodę, a instrukcja `undef` ją usuwa. Słowo kluczowe `undef` może być także używane w klasach (które są tematem rozdziału 7.) do dodefiniowywania ich metod obiektowych. Ciekawe, że za pomocą słowa kluczowego `undef` można oddefiniować metody oddziedziczone bez wpływu na ich definicje w klasach, po których zostały oddziedziczone. Założmy, że klasa `A` zawiera definicję metody `m`, a klasa `B` jest podklassą klasa `A`, dzięki czemu dziedziczy po niej metodę `m` (podklasy i dziedziczenie zostały również opisane w rozdziale 7.). Aby uniemożliwić egzemplarzom klasy `B` wywoływanie metody `m`, można metodę tę oddefiniować w ciele tej podklasy za pomocą słowa kluczowego `undef`.

Instrukcja `undef` nie jest często używana. W praktyce znacznie częściej przeddefiniowuje się metody za pomocą nowej instrukcji `def`, niż **oddefiniowuje**, czyli usuwa je całkowicie.

Po instrukcji `undef` musi znajdować się jeden identyfikator określający nazwę metody. Nie może on zostać użyty do oddefiniowania metody singletonowej w taki sam sposób, w jaki instrukcja `def` do jej zdefiniowania.

W obrębie klasy lub modułu do usuwania definicji metod można także używać metody `undef_method` (prywatnej metody klasy `Module`). Należy do niej przekazać symbol reprezentujący nazwę metody, która ma zostać usunięta.

6.2. Nazwy metod

Zgodnie z konwencją nazwy metod zaczynają się od małej litery (nazwa metody może zaczywać się od wielkiej litery, ale wtedy wygląda jak stała). Jeśli nazwa metody składa się z więcej niż jednego słowa, poszczególne człony tej nazwy są zazwyczaj łączone znakiem podkreślenia `jak_tutaj`, a nie `jakTutaj`.

Rozwijanie nazw metod

Niniejszy podrozdział opisuje nazwy nadawane metodom podczas ich definiowania. Po-krewnym tematem jest rozwijanie nazw metod, czyli sposób, w jaki interpreter znajduje definicję metody, której nazwa została użyta w wyrażeniu wywołania. Odpowiedź na to pytanie musi poczekać, aż poznasz klasy, czyli do podrozdziału 7.8.

Nazwa metody może, ale nie musi, kończyć się znakiem równości, znakiem zapytania lub wykrzyknikiem. Znak równości oznacza, że metoda jest tzw. setterem (metodą ustawiającą) i może być wywoływana przy użyciu składni przypisania. Metody ustawiające zostały opisane w podrozdziale 4.5.3, a dodatkowe ich przykłady można znaleźć w podrozdziale 7.1.5. Znak zapytania i wykrzyknik nie mają żadnego specjalnego znaczenia dla interpretera, ale są dozwolone, ponieważ umożliwiają stosowanie dwóch niezwykle przydatnych konwencji nazewniczych.

Pierwsza z tych konwencji polega na tym, że każda metoda, której nazwa kończy się znakiem zapytania, zwraca wartość odpowiadającą na pytanie postawione w jej wywołaniu. Na przykład metoda `empty?` wywołana na rzecz tablicy zwraca wartość `true`, jeśli tablica ta nie zawiera żadnych elementów. Metody tego typu nazywają się predykatami i zazwyczaj zwracają jedną z wartości logicznych — `true` lub `false` — chociaż nie jest to wymagane, ponieważ każda wartość inna niż `false` i `nil` jest traktowana jako `true`, kiedy wymagana jest wartość logiczna (na przykład metoda klasy `Numeric` o nazwie `nonzero?` zwraca wartość `nil`, jeśli liczba, na rzecz której została wywołana, to zero, lub w przeciwnym przypadku zwraca tę liczbę).

Druga konwencja polega na tym, że przy używaniu metod zakończonych wykrzyknikiem należy zachować szczególną ostrożność. Na przykład obiekty klasy `Array` udostępniają metodę `sort`, która robi kopię tablicy i ją sortuje. Dostępna jest też metoda `sort!` sortująca oryginalną tablicę. Wykrzyknik oznacza, że przy używaniu tej wersji niniejszej metody należy zachować większą ostrożność.

Metody zakończone wykrzyknikiem są często tzw. **mutatorami** (ang. *mutator*), czyli metodami modyfikującymi, ponieważ zmieniają wewnętrzny stan obiektów. Jednak nie zawsze tak jest. Jest wiele metod modyfikujących (mutatorów) niekończących się wykrzyknikiem oraz kilka metod niezmieniających wewnętrznego stanu obiektów, które na końcu mają wykrzykniki. Metody modyfikujące (jak `Array.fill`) nieposiadające niemodyfikującego odpowiednika zazwyczaj nie mają w nazwie wykrzyknika.

Weźmy globalną funkcję `exit`, która zatrzymuje program Ruby w kontrolowany sposób. Istnieje wersja tej metody o nazwie `exit!`, która zamyka program natychmiast po jej wywołaniu, nie wykonując żadnych bloków `END` ani kodu zarejestrowanego za pomocą metody `at_exit`. Metoda `exit!` nie jest mutatorem, tylko „niebezpieczną” wersją metody `exit`. Znak `!` przy pomina programistę, aby używała jej z rozwagą.

6.2.1. Metody operatorowe

Wiele operatorów języka Ruby, na przykład `+`, `*`, a nawet operator indeksów tablicowych `[]`, zostało zaimplementowanych jako metody, które można definiować we własnych klasach. Operator definiuje się poprzez zdefiniowanie metody o takiej samej „nazwie” jak on sam (jedynym wyjątkiem od tej reguły są jednoargumentowe operatory dodawania i odejmowania, których metody mają nazwy `+@` i `-@`). Metodę można zdefiniować, nawet jeśli jej nazwa składa się z samych znaków interpunkcyjnych. W związku z tym można napisać definicję metody jak ta poniżej:

```
def +(other)          # Definicja dwuargumentowego operatora dodawania: x+y równa się x.+(@).  
  self.concatenate(other)  
end
```

Informacje na temat tego, które operatory w języku Ruby są zdefiniowane jako metody, można znaleźć w tabeli 4.2 w rozdziale 4. Operatory te są jedynymi metodami, których nazwy mogą składać się z samych znaków interpunkcyjnych — nie można tworzyć nowych operatorów ani definiować metod, których nazwy składają się tylko ze znaków interpunkcyjnych. Dodatkowe przykłady definicji operatorów opartych na metodach można znaleźć w podrozdziale 7.1.6.

Metody definiujące jednoargumentowe operatory nie przyjmują żadnych argumentów. Metody definiujące operatory dwuargumentowe przyjmują jeden argument i powinny działać na obiekcie `self` i tym argumentem. Operatory dostępu do tablicy `[]` i `[]=` są wyjątkowe, ponieważ można je wywoływać z dowolną liczbą argumentów. Dla operatora `[]=` ostatni argument jest zawsze tą wartością, która jest przypisywana.

6.2.2. Aliasy metod

W języku Ruby wiele metod ma więcej niż jedną nazwę. Do definiowania nowych nazw dla istniejących już metod służy słowo kluczowe `alias`. Oto przykład jego użycia:

```
alias aka also_known_as  # alias nowa_nazwa_stara_nazwa.
```

Po wykonaniu tej instrukcji identyfikator `aka` będzie odnosił się do tej samej metody co identyfikator `also_known_as`.

Możliwość tworzenia aliasów nazw metod sprawia, że język Ruby jest bardzo ekspresywny i naturalny. Kiedy jedna metoda ma kilka nazw, można wybrać tę najlepiej pasującą do określonego kontekstu. Na przykład klasa `Range` zawiera metodę służącą do sprawdzania, czy określona wartość mieści się w danym przedziale. Można ją wywołać za pomocą identyfikatora `include?` lub `member?`. Jeśli przedział potraktujesz jako rodzaj zbioru, najlepszym wyborem wydaje się nazwa `member?`.

Bardziej praktycznym zastosowaniem aliasów metod jest nadawanie metodom nowych funkcji. Poniżej znajduje się przykład wzbogacania istniejącej metody:

```

def hello
  puts "Witaj świecie"
end
alias original_hello hello
def hello
  puts "Proszę o uwagę"
  original_hello
  puts "To były ćwiczenia"
end
# Fajna prosta metoda.
# Założmy, że chcesz ją nieco wzbogacić...
# Nadajesz metodzie zapasową nazwę.
# Definiujesz nową metodę ze starej nazwą.
# Metoda coś robi.
# Następnie wywołuje oryginalną metodę.
# Metoda robi coś jeszcze.

```

W powyższym kodzie wykorzystano metody globalne. Częściej słowo kluczowe alias jest używane do tworzenia nowych nazw dla metod klasowych (więcej na ich temat dowiesz się w rozdziale 7.). W takiej sytuacji słowo kluczowe alias musi znajdować się wewnątrz definicji klasy, której metoda ma mieć zmienioną nazwę. W języku Ruby możliwe jest otwieranie klas (również opisane w rozdziale 7.), co oznacza, że można istniejącą klasę otworzyć w swoim kodzie za pomocą instrukcji `class` i użyć słowa kluczowego `alias`, tak jak w powyższym przykładzie, do wzbogacenia lub modyfikacji metod znajdujących się w tej klasie. Nazywa się to „tworzeniem łańcuchów aliasów” i zostało opisane w podrozdziale 8.11.

Tworzenie aliasów to nie przeciążanie

W języku Ruby jedna metoda może mieć dwie nazwy, ale dwie metody nie mogą mieć jednej nazwy. W językach typowanych statycznie metody można rozróżnić na podstawie liczby i typu przyjmowanych przez nie argumentów. Dzięki temu jedna lub więcej metod może mieć taką samą nazwę, jeśli przyjmują różne liczby lub typy argumentów. Takie przeciążanie nie jest możliwe w języku Ruby.

Z drugiej strony przeciążanie metod nie jest w języku Ruby w ogóle potrzebne. Metody mogą przyjmować argumenty dowolnej klasy i można je tak zaimplementować, aby wykonywały różne działania w zależności od przyjętych argumentów. Ponadto (jak zobaczysz później) możliwe jest deklarowanie domyślnych wartości dla argumentów metod. Argumenty te mogą być pomijane w wywołaniach metod. To pozwala na wywoływanie jednej metody z różną liczbą argumentów.

6.3. Nawiasy w metodach

W większości wywołań metod nawiasy można opuścić. W prostych przypadkach pozwala to na napisanie czystszego kodu. Jednak w przypadkach bardziej skomplikowanych powoduje to dwuznaczności syntaktyczne i trudne do zrozumienia sytuacje bez wyjścia. Opisane one zostały w poniższych podrozdziałach.

6.3.1. Opcjonalne nawiasy

Nawiasy w wywołaniach metod w języku Ruby są bardzo często opuszczane. Na przykład oba poniższe wiersze kodu są sobie równoważne:

```

puts "Witaj świecie"
puts("Witaj świecie")

```

W pierwszym wierszu `puts` wygląda jak słowo kluczowe, instrukcja lub wbudowane w język polecenie. Drugi wiersz jasno pokazuje, że jest to wywołanie globalnej metody z opuszczonymi

nawiasami. Mimo iż druga z tych wersji jest bardziej przejrzysta, pierwsza jest bardziej zwięzła, częściej używana i prawdopodobnie bardziej naturalna.

Teraz spójrz na poniższy fragment kodu:

```
greeting = "Hello"  
size = greeting.length
```

Osoby przyzwyczajone do innych obiektowych języków programowania mogą pomyśleć, że `length` to własność, pole lub zmienna zawarta w obiektach łańcuchów. Jednak język Ruby jest ściśle obiektowy, a więc jego obiekty są w pełni hermetyczne. Jedyna droga do nich prowadzi poprzez ich metody. W powyższym kodzie `greeting.length` jest wywołaniem metody. Metoda `length` nie pobiera żadnych argumentów i została wywołana bez nawiasów. Poniższy kod jest równoznaczny z powyższym:

```
size = greeting.length()
```

Dodanie nawiasów podkreśla fakt, że wywoływana jest metoda. Bardzo często praktykowane opuszczenie nawiasów w wywołaniach metod, które nie przyjmują żadnych argumentów, sprawia wrażenie, że jest to instrukcja dostępu do własności.

Nawiasy opuszcza się bardzo często wówczas, gdy wywoływana metoda przyjmuje zero argumentów lub jeden argument. Nawiasy można opuścić, nawet jeśli metoda przyjmuje więcej niż jeden argument, ale jest to rzadziej wykorzystywana możliwość. Oto przykład:

```
x = 3          # x jest liczbą.  
x.between? 1,5    # To samo co x.between?(1,5).
```

Można także opuścić nawiasy otaczające listy parametrów w definicjach metod, ale trudno twierdzić, że dzięki temu kod staje się bardziej przejrzysty lub czytelny. Na przykład poniższa procedura definiuje metodę zwracającą sumę swoich argumentów:

```
def sum x, y  
  x+y  
end
```

6.3.2. Obowiązkowe używanie nawiasów

Czasami opuszczenie nawiasów powoduje, że kod staje się niejednoznaczny. W takich sytuacjach użycie nawiasów jest obowiązkowe. Najczęstszym przypadkiem tego typu są zagnieżdżone wywołania metod w rodzaju `f g x, y`. W Ruby takie wywołanie jest równoznaczne z wywołaniem `f(g(x, y))`. Jednak Ruby 1.8 zgłasza w takiej sytuacji ostrzeżenie, ponieważ kod ten można również zinterpretować jako `f(g(x), y)`. Ostrzeżenie to nie jest już zgłasiane w Ruby 1.9. Poniższy kod wykorzystuje zdefiniowaną powyżej metodę `sum` i drukuje wartość 4, ale w Ruby 1.8 zgłasza ostrzeżenie:

```
puts sum 2,2
```

Aby pozbyć się ostrzeżenia, należy argumenty umieścić w nawiasach:

```
puts sum(2,2)
```

Warto zauważyc, że użycie nawiasów dla zewnętrznego wywołania metody nie rozwiązuje problemu dwuznaczności:

```
puts(sum 2,2)  # Znaczy puts(sum(2,2)) czy puts(sum(2), 2)?
```

Wyrażenie zawierające zagnieździone wywołania funkcji jest dwuznaczne tylko wtedy, gdy w grę wchodzi więcej niż jeden argument. Poniższy kod może zostać zinterpretowany przez interpreter tylko w jeden sposób:

```
puts factorial x  # To może oznaczać tylko puts(factorial(x)).
```

Mimo braku dwuznaczności w tym przypadku Ruby 1.8 i tak zgłosi ostrzeżenie, jeśli zostaną opuszczone nawiasy wokół `x`.

Czasami opuszczenie nawiasów oznacza prawdziwy błąd składni. Na przykład poniższe instrukcje są bez nawiasów kompletnie niejednoznaczne, przez co interpreter nawet nie próbuje zgadnąć, co programista miał na myśli:

```
puts 4, sum 2,2  # Błąd: drugi przecinek należy do pierwszej czy drugiej metody?  
[sum 2,2]        # Błąd: dwa elementy tablicowe czy jeden?
```

Możliwość opuszczania nawiasów powoduje jeszcze jeden problem. Jeśli w wywołaniu metody **nawiasy** zostaną użyte, otwierający nawias **musi** znajdować się bezpośrednio za nazwą metody, bez żadnych białych znaków pomiędzy nimi. Konieczność ta wynika z podwójnej funkcji nawiasów — mogą one otaczać listę argumentów w wywołaniu metody i grupować wyrażenia. Przeanalizuj poniższe dwa wyrażenia, które różnią się tylko jedną spacją:

```
square(2+2)*2    # square(4)*2 = 16*2 = 32  
square (2+2)*2   # square(4*2) = square(8) = 64
```

W pierwszym z tych wyrażeń nawiasy reprezentują wywołanie metody. W drugim natomiast służą grupowaniu wyrażeń. Aby zmniejszyć ryzyko potencjalnej pomyłki, należy zawsze stosować nawiasy wokół wywołań metod, jeśli którykolwiek z jej argumentów wymaga użycia nawiasów. Drugie wyrażenie można zapisać następująco:

```
square((2+2)*2)
```

Na zakończenie tematu nawiasów przedstawimy jeszcze jeden ciekawy przypadek. Przypomnijmy, że poniższe wyrażenie jest dwuznaczne i powoduje ostrzeżenie:

```
puts(sum 2,2)  # To znaczy puts(sum(2,2)) czy puts(sum(2), 2)?
```

Najlepszym sposobem na uniknięcie dwuznaczności jest w tym przypadku wstawienie argumentów metody `sum` w nawiasy. Innym rozwiązaniem jest wstawienie spacji pomiędzy identyfikatorem `puts` a nawiasem otwierającym:

```
puts (sum 2,2)
```

Dodanie spacji spowodowało zmianę funkcji nawiasów wywołania metody na nawiasy grupujące wyrażenie. Ponieważ nawiasy te grupują podwyrażenie, przecinek nie może już być interpretowany jako znak rozdzielający argumenty wywołania metody `puts`.

6.4. Argumenty metod

W prostych deklaracjach metod po nazwie metody znajduje się lista argumentów rozdzielonych przecinkami (opcjonalnie w nawiasach). Jednak o argumentach metod w języku Ruby można powiedzieć znacznie więcej. Kolejne podrozdziały opisują:

- Sposoby deklarowania argumentów z wartościami domyślnymi, dzięki czemu można je pomijać przy wywoływaniu metody.
- Sposoby deklaracji metod przyjmujących dowolną liczbę argumentów.

- Sposoby symulowania wyznaczonych argumentów metod przy użyciu specjalnej składni, aby móc przekazywać do metod tablice asocjacyjne.
- Deklarowanie metod w taki sposób, aby blok związany z ich wywołaniami był traktowany jako ich argument.

6.4.1. Domyślne wartości parametrów

Definiując metodę, można określić wartości domyślne dla niektórych lub wszystkich jej parametrów. Jeśli się to zrobi, metodę taką będzie można wywoływać przy użyciu mniejszej liczby argumentów niż zadeklarowana liczba parametrów. Jeśli jakieś argumenty zostaną opuszczone, zostanie użyta wartość domyślna dla odpowiednich parametrów. Wartości domyślne dla parametrów należy podawać po nazwach tych parametrów, po znaku równości:

```
def prefix(s, len=1)
  s[0,len]
end
```

Niniejsza metoda deklaruje dwa parametry, z których drugi ma wartość domyślną. Oznacza to, że w jej wywołaniu można podać jeden lub dwa argumenty:

```
prefix("Ruby", 3)      # => "Rub"
prefix("Ruby")         # => "R"
```

Wartości domyślne argumentów nie muszą być stałe — mogą to być dowolne wyrażenia, mogą też odwoływać się do zmiennych obiektowych i wcześniejszych parametrów z listy parametrów. Na przykład:

```
# Zwraca ostatni znak s lub podciąg zaczynający się w miejscu index i trwający do końca.
def suffix(s, index=s.size-1)
  s[index, s.size-index]
end
```

Wartości parametrów są obliczane podczas wywoływania metody, a nie podczas jej analizy przez interpreter. W poniższym przykładzie wartość domyślna [] tworzy nową pustą tablicę przy każdym wywołaniu, a nie używa tej samej tablicy utworzonej przy definicji metody:

```
# Dodanie wartości x do tablicy a, zwrócenie a.
# Jeżeli nie została podana żadna tablica, zostaje utworzona pusta tablica.
def append(x, a=[])
  a << x
end
```

W Ruby 1.8 parametry z wartościami domyślnymi muszą znajdować się za wszystkimi zwykłymi parametrami na liście. W Ruby 1.9 zasady te zostały rozluźnione i parametry z wartościami domyślnymi mogą znajdować się przed zwykłymi parametrami. Nadal jednak wszystkie parametry mające wartości domyślne muszą znajdować się obok siebie — na przykład pomiędzy dwoma parametrami mającymi wartości domyślne nie można wstawić parametru bez wartości domyślnej. Jeśli metoda ma więcej niż jeden parametr z wartością domyślną i zostanie wywołana z argumentami dla niektórych z nich, ale nie dla wszystkich, wartości są im przydzielańskie od lewej do prawej. Założmy, że metoda ma dwa parametry z wartościami domyślnymi. Można ją wywołać bez argumentów, z jednym argumentem lub dwoma argumentami. Jeśli zostanie podany jeden argument, zostanie on przypisany do pierwszego parametru, a drugi będzie miał wartość domyślną. Nie ma natomiast możliwości podania wartości dla drugiego parametru przy zachowaniu wartości domyślnej pierwszego.

6.4.2. Nieustalona liczba argumentów i tablice

Czasami potrzebna jest metoda, która może przyjmować dowolną liczbę argumentów. Aby taką utworzyć, należy przed jednym z jej parametrów postawić znak *. W ciele tej metody niniejszy parametr będzie odnosił się do tablicy zawierającej zero lub więcej argumentów przekazanych na tej pozycji. Na przykład:

```
# Zwraca największy z przekazanych argumentów.
def max(first, *rest)
  # Założenie, że pierwszy wymagany argument jest największy.
  max = first
  # Iteracja przez wszystkie opcjonalne argumenty w celu sprawdzenia, czy któryś jest większy.
  rest.each { |x| max = x if x > max }
  # Zwrócenie największego znalezionego argumentu.
  max
end
```

Metoda `max` wymaga podania przynajmniej jednego argumentu, ale może przyjąć dowolną liczbę dodatkowych argumentów. Pierwszy argument jest dostępny poprzez parametr `first`. Pozostałe opcjonalne argumenty są przechowywane w tablicy `rest`. Metodę `max` można wywołać następująco:

```
max(1)      #first=1, rest=[]
max(1, 2)    #first=1, rest=[2].
max(1, 2, 3) #first=1, rest=[2,3].
```

Warto zauważyć, że w Ruby wszystkie obiekty `Enumerable` automatycznie dostają metodę `max`, dlatego zdefiniowana tutaj metoda nie należy do najbardziej przydatnych.

Prefiksem * można opatrzyć najwyżej jeden parametr. W Ruby 1.8 parametr ten musi znajdować się za wszystkimi zwykłymi parametrami i za wszystkimi parametrami mającymi określoną wartość domyślną. Powinien to być ostatni parametr metody, chyba że jest jeszcze parametr z prefiksem & (zobacz poniżej). W Ruby 1.9 parametr z prefiksem * również musi znajdować się za wszystkimi parametrami mającymi wartości domyślne, ale mogą znajdować się za nim zwykłe parametry. Ponadto nadal musi znajdować się przed parametrami z prefiksem &.

6.4.2.1. Przekazywanie tablic do metod

Wiadomo już, jak za pomocą gwiazdki użytej w deklaracji metody spowodować zgromadzenie kilku argumentów w jednej tablicy. Gwiazdki można także użyć w wywołaniu metody do rozbicia tablicy (lub przedziału albo enumeratora) na elementy w taki sposób, aby każdy jej element stał się odrębnym argumentem metody. Gwiazdka jest czasami nazywana operatorem splat, mimo iż nie jest prawdziwym operatorem. Przykłady jej użycia przedstawione zostały w podrozdziale 4.5.5 opisującym przypisywanie równoległe.

Załóżmy, że chcesz znaleźć największą wartość w tablicy (i nie wiesz, że tablice w Ruby posiadają wbudowaną metodę `max`). Można by przekazać elementy tej tablicy do metody `max` (zdefiniowanej wcześniej) w następujący sposób:

```
data = [3, 2, 1]
m = max(*data)  #first = 3, rest=[2,1] => 3.
```

Zobacz, co by się stało, gdyby usunięto gwiazdkę:

```
m = max(data)  #first = [3,2,1], rest=[] => [3,2,1].
```

W tym przypadku tablica jest przekazywana jako pierwszy i jedyны argument, a metoda `max` zwraca go, nie wykonując żadnych operacji porównywania.

Gwiazdki można także użyć do rozbijania zwracanych przez metody tablic, aby można je było używać w wywołaniach innych metod. Weźmy na przykład zdefiniowane wcześniej metody polar i cartesian:

```
# Konwersja punktu (x,y) na współrzędne biegunowe i z powrotem na kartezjańskie.  
x, y = cartesian(*polar(x, y))
```

W Ruby 1.9 enumeratory są obiektami, do których można stosować gwiazdkę. Na przykład aby znaleźć największą literę włańcucha, można napisać:

```
max(*"witaj".each_char) #=> 'w'
```

6.4.3. Mapowanie argumentów na parametry

Jeśli w definicji metody znajdują się parametry z wartościami domyślnymi lub parametr z gwiazdką, przypisywanie wartości do parametrów podczas wywoływania metody staje się nieco skomplikowane.

W Ruby 1.8 położenie specjalnych parametrów jest ściśle określone, dzięki czemu wartości argumentów są przypisywane parametrom od lewej do prawej. Pierwsze argumenty są przypisywane do zwykłych parametrów. Jeśli zostają jeszcze jakieś argumenty, są one przypisywane do parametrów mających wartości domyślne. Jeżeli mimo to nadal pozostają jakieś argumenty, zostają one przypisane do argumentu tablicowego.

W Ruby 1.9 konieczne było zastosowanie sprytniejszego algorytmu mapowania argumentów na parametry, ponieważ kolejność parametrów nie podlega już takim obostrzeniom. Założymy, że masz metodę, w której deklaracji znajduje się o zwykłych parametrów, d parametrów z wartościami domyślnymi i jedna tablica z prefiksem *, oraz że parametry te występują w dowolnym porządku. Założymy teraz, że wywołujesz tę metodę z liczbą argumentów.

W sytuacji gdy a jest mniejsza od o, zostaje zgłoszony wyjątek ArgumentError, ponieważ nie podano minimalnej wymaganej liczby argumentów.

Jeśli a jest większa lub równa o i mniejsza lub równa o+d, to argumenty zostaną przypisane do a-o, licząc od lewej parametrów mających wartości domyślne. Pozostałe (do prawej) o+d-a parametrów będzie miało swoje wartości domyślne.

Gdy a jest większa od o+d, w parametrze tablicowym mającym prefiks * zostanie zapisanych a-o-d argumentów. W przeciwnym razie parametr ten będzie pusty.

Po wykonaniu tych obliczeń argumenty są mapowane na parametry od lewej do prawej. Każdemu parametrowi jest przypisywana odpowiednia liczba argumentów.

6.4.4. Tablice asocjacyjne

Jeżeli metoda pobiera dwa lub trzy argumenty, programiecie może być trudno zapamiętać właściwą kolejność w wywołaniu. Niektóre języki programowania pozwalają na pisanie wywołań metod, które jawnie określają nazwę parametru dla każdego przekazywanego argumentu. Ruby nie obsługuje tej składni wywoływania metod, ale można uzyskać podobny efekt, pisząc metodę przyjmującą jako argument lub jeden z argumentów tablicę asocjacyjną:

```
# Niniejsza metoda zwraca tablicę n liczb. Dla każdego indeksu i, 0 <= i < n  
# wartość elementu a[i] wynosi m*i+c. Argumenty n, m i c są przekazywane jako  
# klucze w tablicy asocjacyjnej, dzięki czemu nie trzeba pamiętać ich kolejności.  
def sequence(args)
```

```

# Pobranie argumentów z tablicy asocjacyjnej.
# Zwróć uwagę na użycie operatora || do określenia wartości domyślnych używanych,
# jeśli tablica asocjacyjna nie zawiera szukanego klucza.
n = args[:n] || 0
m = args[:m] || 1
c = args[:c] || 0
a = []                                # Pusta tablica.
n.times { |i| a << m*i+c }      # Obliczenie wartości każdego elementu tablicy.
a                                         # Zwrócenie tablicy.
end

```

Metodę tę można wywołać, podając do niej jako argument literał haszowy:

```
sequence({ :n=>3, :m=>5 })      # => [0, 5, 10].
```

Aby uprościć ten styl programowania, Ruby pozwala na opuszczenie nawiasów klamrowych z literałem haszowym, jeśli jest on ostatnim argumentem metody (lub jeśli po nim znajduje się tylko argument blokowy poprzedzony prefiksem &). Tablica asocjacyjna bez klamer wygląda tak, jakby przekazywane były osobne argumenty, które można sortować w dowolny sposób:

```
sequence(:m=>3, :n=>5)          # => [0, 3, 6, 9, 12].
```

Podobnie jak w innych metodach Ruby, tutaj również można opuścić nawiasy:

```
# Składnia Ruby 1.9.
sequence c:1, m:3, n:5           # => [1, 4, 7, 10, 13].
```

Jeśli zostaną opuszczone nawiasy, **muszą** zostać opuszczone klamry. W przypadku gdy klamry znajdują się za nazwą metody są poza nawiasami, interpreter przyjmuje, że do metody jest przekazywany blok:

```
sequence { :m=>3, :n=>5 }        # Blqd skłdn!
```

6.4.5. Argumenty blokowe

Przypomnijmy z podrozdziału 5.3, że blok to fragment kodu związany z wywołaniem metody oraz że iterator to metoda, która takiego bloku wymaga. Blok może znajdować się za wywołaniem każdej metody oraz każda metoda, po której znajduje się blok kodu, może wywołać ten kod za pomocą instrukcji yield. Aby odświeżyć pamięć, poniższy kod przedstawia przyjmującą blok wersję metody sequence, która została utworzona wcześniej w niniejszym rozdziale:

```

# Generuje szereg n liczb m*i + c oraz przekazuje je do bloku.
def sequence2(n, m, c)
  i = 0
  while(i < n)           # Powtóżenie n razy.
    yield i*m + c         # Przekazanie kolejnego elementu szeregu do bloku.
    i += 1
  end
end
# Przykładowy sposób użycia powyższej metody.
sequence2(5, 2, 2) { |x| puts x } # Drukuje liczby 2, 4, 6, 8, 10.

```

Jedną z cech szczególnych bloków jest ich anonimowość. Nie są przekazywane do metod w tradycyjny sposób, nie posiadają nazw i są wywoływanie za pomocą słowa kluczowego, a nie metody. Aby zyskać bardziej rzeczywistą kontrolę nad blokiem (na przykład by móc przekazać go do jakieś innej metody), należy przed ostatnim argumentem swojej metody postawić prefiks &¹.

¹ Parametry metod z prefiksem & nazywamy „argumentami blokowymi” zamiast „parametrami blokowymi”, ponieważ termin „parametr blokowy” odnosi się do listy parametrów (na przykład |x|) samego bloku.

Wtedy argument ten będzie odnosił się do bloku — jeśli istnieje — przekazywanego do tej metody. Wartością tego argumentu będzie obiekt klasy Proc, a więc zamiast używać instrukcji yield, należy wywołać metodę call klasy Proc:

```
def sequence3(n, m, c, &b) # Jawnym argumentem pozwalającym zmienić blok na obiekt klasy Proc.  
  i = 0  
  while(i < n)  
    b.call(i*m + c)          # Wywołanie obiektu klasy Proc za pomocą jego metody call.  
    i += 1  
  end  
end  
# Blok jest nadal przekazywany poza nawiasami.  
sequence3(5, 2, 2) { |x| puts x }
```

Użycie znaku & w ten sposób zmienia tylko definicję metody. Wywołanie jej pozostaje niezmienione. Otrzymujesz argument blokowy zadeklarowany w nawiasach definicji metody, przy czym sam blok nadal pozostaje poza nawiasami jej wywołania.

Jawne przekazywanie obiektów klasy Proc

Mając utworzony obiekt klasy Proc (jak go utworzyć, nauczysz się dalej) i chcąc go jawnie przekazać do jakiejś metody, możesz to zrobić tak samo jak z każdą inną wartością — obiekt klasy Proc jest takim samym obiektem jak wszystkie inne. W takim przypadku nie należy używać znaku & w definicji metody:

```
# Niniejsza wersja przyjmuje jawnie utworzony obiekt klasy Proc, nie blok.  
def sequence4(n, m, c, b) # Argument b nie ma prefiksu &.  
  i = 0  
  while(i < n)  
    b.call(i*m + c)      # Jawne wywołanie obiektu klasy Proc.  
    i += 1  
  end  
end  
p = Proc.new { |x| puts x } # Jawnie utworzenie obiektu klasy Proc.  
sequence4(5, 2, 2, p)      # Przekazanie go jako zwykłego argumentu.
```

Dwa razy już napisaliśmy w tym rozdziale, że jakiś specjalny rodzaj parametru musi znajdować się na samym końcu listy parametrów. Argumenty blokowe z prefiksem & **naprawdę** muszą występować jako ostatnie. Ponieważ bloki są w wywołaniach metod przekazywane w nietypowy sposób, nazwane argumenty blokowe są inne i nie oddziałują z parametrami tablicowymi i haszowymi, w których opuszczono nawiasy i klamry. Na przykład poniższe dwie metody są poprawne:

```
def sequence5(args, &b) # Przekazanie argumentów jako tablicy haszowej i dodanie bloku.  
  n, m, c = args[:n], args[:m], args[:c]  
  i = 0  
  while(i < n)  
    b.call(i*m + c)  
    i += 1  
  end  
end  
# Przyjmuje jeden lub więcej argumentów, po których następuje blok.  
def max(first, *rest, &block)  
  max = first  
  rest.each { |x| max = x if x > max }  
  block.call(max)  
  max  
end
```

Metody te działają, ale można je uprościć, pozostawiając bloki anonimowymi i wywołując je za pomocą instrukcji `yield`.

Ponadto warto zauważyć, że instrukcja `yield` działa także w metodzie, w której definicji znajduje się parametr z prefiksem `&`. Nawet jeśli blok został przekonwertowany na klasę `Proc` i przekazany jako argument, nadal można go wywołać jako blok anonimowy, tak jakby argumentu blokowego nie było.

6.4.5.1. Znak `&` w wywołaniach metod

Wiadomo już, że znak `*` w definicji metody powoduje spakowanie kilku argumentów do tablicy, a w wywołaniu metody rozpakowanie tablicy, tak że jej poszczególne elementy są osobnymi argumentami. Znaku `&` również można używać w definicjach i wywołaniach. Wcześniej dowiedziałeś się, że znak `&` w definicji metody pozwala na użycie zwykłego bloku związanego z wywołaniem tej metody jako nazwanego obiektu klasy `Proc` wewnątrz tej metody. Jeżeli w wywołaniu metody przed obiektem klasy `Proc` znajdzie się znak `&`, obiekt ten jest traktowany przez tę metodę jako zwykły blok znajdujący się za wywołaniem.

Przeanalizuj poniższy fragment kodu, który sumuje zawartość dwóch tablic:

```
a, b = [1,2,3], [4,5]          # Jakiś dane początkowe.  
sum = a.inject(0) { |total,x| total+x }    # => 6. Suma elementów tablicy a.  
sum = b.inject(sum) { |total,x| total+x }   # => 15. Dodanie elementów z tablicy b.
```

W podrozdziale 5.3.2 została opisana metoda `inject`. Aby przypomnieć sobie jej zastosowanie, można znaleźć ją w dokumentacji za pomocą polecenia `ri Enumerable.inject`. W powyższym kodzie na uwagę zasługuje to, że oba użyte w nim bloki są identyczne. Zamiast zmuszać interpreter do analizy dwa razy tego samego bloku, można utworzyć obiekt klasy `Proc` reprezentujący ten blok i użyć go dwa razy:

```
a, b = [1,2,3], [4,5]          # Jakiś dane początkowe.  
summation = Proc.new { |total,x| total+x } # Obiekt klasy Proc.  
sum = a.inject(0, &summation)           # => 6.  
sum = b.inject(sum, &summation)         # => 15.
```

Znak `&` w wywołaniu metody musi znajdować się przed ostatnim argumentem tego wywołania. Blok może być związany z każdym wywołaniem metody, nawet takiej, która nie oczekuje bloku i nie używa instrukcji `yield`. Podobnie w każdym wywołaniu metody można użyć znaku `&` przed ostatnim argumentem.

W wywołaniach metod znak `&` zazwyczaj znajduje się przed obiektem klasy `Proc`, chociaż może zostać użyty przed dowolnym obiektem udostępniającym metodę `to_proc`. Metodę tę definiuje klasa `Method` (opisana dalej w niniejszym rozdziale), dlatego obiekty tej klasy mogą być przekazywane do iteratorów tak jak obiekty klasy `Proc`.

W Ruby 1.9 metodę `to_proc` zawiera klasa `Symbol`, dzięki czemu prefiks `&` można stawiać przed symbolami oraz przekazywać je do iteratorów. Symbol przekazywany w ten sposób jest traktowany jako nazwa metody. Obiekt klasy `Proc` zwrócony przez metodę `to_proc` wywołuje tak nazwaną metodę swojego pierwszego argumentu i przekazuje do niej wszystkie swoje pozostałe argumenty. Oto typowy przykład: mając tablicę łańcuchów, utwórz nową tablicę tych łańcuchów przekonwertowanych na wielkie litery. Metoda `Symbol.to_proc` pozwala na zgrabne wykonanie tego zadania:

```
words = ['oraz', 'but', 'kot']      # Tablica słów.  
uppercase = words.map &:upcase     # Konwersja na wielkie litery za pomocą metody String.upcase.  
upper = words.map { |w| w.upcase } # Ten sam kod zapisany z użyciem bloku.
```

6.5. Obiekty proc i lambda

Bloki w języku Ruby są strukturami syntaktycznymi. Nie są obiektami, a więc nie można wykonywać na nich takich operacji jak na obiektach. Można natomiast utworzyć obiekt reprezentujący blok. W zależności od sposobu utworzenia obiekt ten to **proc** lub **lambda**. Obiekty proc zachowują się jak bloki, a lambdy jak metody. Jedne i drugie są natomiast egzemplarzami klasy Proc.

Poniższe podrozdziały opisują:

- tworzenie obiektów klasy Proc w formie proc i lambda;
- wywoływanie obiektów klasy Proc;
- sprawdzanie, ilu argumentów wymaga obiekt klasy Proc;
- sposoby sprawdzania, czy dwa obiekty klasy Proc są takie same;
- różnice pomiędzy obiektami proc a lambdami.

6.5.1. Tworzenie obiektów klasy Proc

Do tej pory przedstawiliśmy jeden sposób tworzenia obiektów klasy Proc polegający na związaniu bloku z metodą, w której definicji znajduje się argument blokowy z przedrostkiem &. Nic nie stoi na przeszkodzie, aby taka metoda zwracała obiekt klasy Proc do użytku poza nią:

```
# Niniejsza metoda tworzy obiekt proc z blokiem.  
def makeproc(&p)  # Konwersja bloku na obiekt klasy Proc i zapisanie go w zmiennej p.  
  p  
    # Zwrócenie obiektu klasy Proc.  
end
```

Mając zdefiniowaną taką metodę, można tworzyć obiekty klasy Proc na własny użytek:

```
adder = makeproc { |x,y| x+y }
```

Zmienna adder odnosi się teraz do obiektu klasy Proc. Obiekty klasy Proc tworzone w ten sposób są obiektami proc, nie lambdami. Wszystkie obiekty klasy Proc udostępniają metodę call, której wywołanie powoduje uruchomienie kodu bloku, z którego obiekt proc został utworzony. Na przykład:

```
sum = adder.call(2,2)  #=> 4
```

Poza wywoływaniem obiekty klasy Proc można przekazywać do metod, zapisywać w strukturach danych i wykonywać na nich działania jak na wszystkich innych obiektach w języku Ruby.

Poza tworzeniem obiektów proc poprzez wywołania metod istnieją trzy metody tworzące obiekty klasy Proc (zarówno proc, jak i lambdy). Metody te są powszechnie używane i definiowanie przedstawionej wcześniej metody makeproc nie jest w rzeczywistości konieczne. Poza tymi metodami w Ruby 1.9 dostępna jest nowa składnia literałowa pozwalająca definiować lambdy. Kolejne podrozdziały opisują metody Proc.new, lambda i proc oraz składnię literałów lambda w Ruby 1.9.

6.5.1.1. Metoda Proc.new

Metoda `Proc.new` została już użyta w kilku prezentowanych dotychczas w tym rozdziale przykładach. Jest to zwykła metoda `new` dostępna w większości klas. Jej użycie jest najbardziej oczywistym sposobem na utworzenie nowego egzemplarza klasy `Proc`. Metoda ta nie pobiera żadnych argumentów, a zwraca obiekt klasy `Proc` typu `proc` (nie `lambda`). Kiedy zostanie ona wywołana z blokiem, zwraca obiekt `proc` reprezentujący ten blok. Na przykład:

```
p = Proc.new { |x,y| x+y }
```

Jeśli metoda `Proc.new` zostanie wywołana bez bloku wewnętrz metody, z którą jest związany blok, zwraca obiekt `proc` reprezentujący blok tej zewnętrznej metody. Taki sposób użycia `Proc.new` jest alternatywą dla stosowania argumentów blokowych poprzedzonych znakiem `&` w definicjach metod. Na przykład poniższe dwie ustawione obok siebie metody są równoznaczne:

```
def invoke(&b)      def invoke
  b.call            Proc.new.call
end                end
```

6.5.1.2. Metoda Kernel.lambda

Innym sposobem tworzenia obiektów klasy `Proc` jest użycie metody `lambda` z modułu `Kernel`, która zachowuje się jak funkcja globalna. Jak wskazuje nazwa, metoda ta zwraca obiekty klasy `Proc` będące lambdami, a nie obiektami `proc`. Metoda `lambda` nie pobiera żadnych argumentów, ale z jej wywołaniem musi być związany blok kodu:

```
is_positive = lambda { |x| x > 0 }
```

Historia lambdy

Lambdy i metoda `lambda` zawdzięczają swoją nazwę rachunkowi lambda, gałęzi logiki matematycznej, która znalazła zastosowanie w funkcjonalnych językach programowania. W języku Lisp funkcje, którymi można operować jak obiektami, również nazywają się lambdami.

6.5.1.3. Metoda Kernel.proc

W Ruby 1.8 globalna metoda `proc` jest synonimem metody `lambda`. Pomimo nazwy zwraca lambdy, a nie obiekty `proc`. W Ruby 1.9 poprawiono to. W tej wersji języka metoda `proc` jest synonimem metody `Proc.new`.

Ze względu na tę niejednoznaczność metody `proc` nie należy nigdy używać w Ruby 1.8. Działanie kodu mogłoby ulec zmianie, gdyby interpreter został zaktualizowany do nowszej wersji. Pisząc kod Ruby 1.9 i mając pewność, że nigdy nie będzie on uruchamiany za pomocą interpretera Ruby 1.8, można bezpiecznie używać metody `proc` jako eleganckiego zamiennika dla metody `Proc.new`.

6.5.1.4. Literaty lambd

W Ruby 1.9 wprowadzono nową składnię pozwalającą na definiowanie lambd jako literałów. Zaczniemy od lambdy w Ruby 1.8 utworzonej za pomocą metody `lambda`:

```
succ = lambda { |x| x+1}
```

W Ruby 1.9 kod ten można przekonwertować na literal w następujący sposób:

- Zastąp nazwę metody lambda znakami `->`.
- Przenieś listę argumentów przed klamry.
- Listę argumentów otocz nawiasami `()` zamiast znakami `||`.

Po dokonaniu tych zmian powstanie literalu lambdy w Ruby 1.9:

```
succ = ->(x){ x+1 }
```

Zmienna `succ` przechowuje teraz obiekt klasy `Proc`, którego można używać jak każdego innego obiektu:

```
succ.call(2)      #=> 3.
```

Wprowadzenie tej składni do języka Ruby było kontrowersyjne i potrzeba trochę czasu, aby się do niej przyzwyczaić. Należy zauważać, że strzałki `->` są nieco inne niż w literałach haszowych. W literałach lambd używa się strzałek, w których skład wchodzi myślnik, podczas gdy w literałach haszowych strzałka zawiera znak równości.

Podobnie jak w blokach w Ruby 1.9, lista argumentów literala lambdy może zawierać deklaracje zmiennych blokowych, które nie przesłaniają zmiennych o tych samych nazwach dostępnych w wyższym zakresie. Listę takich zmiennych lokalnych należy wpisać po średniku postawionym za listą parametrów:

```
# Należy zwrócić uwagę, że obiekt lambda pobiera dwa argumenty i deklaruje trzy zmienne lokalne.  
f = ->(x,y; i,j,k) { ... }
```

Jedną z zalet tej nowej składni w stosunku do technik tworzenia lambd z bloków jest to, że pozwala ona na deklarowanie lambd (podobnie jak metod) z argumentami posiadającymi wartości domyślne:

```
zoom = ->(x,y,factor=2) { [x*factor, y*factor] }
```

Tak samo jak w deklaracjach metod nawiasy w literałach lambd są opcjonalne, ponieważ lista parametrów i listy zmiennych lokalnych są oddzielone strzałkami `->` oraz znakami `; i {`. Trzy powyższe lambdy można zatem zapisać następująco:

```
succ = ->x { x+1 }  
f = ->x,y; i,j,k { ... }  
zoom = ->x,y,factor=2 { [x*factor, y*factor] }
```

Parametry lambd i zmienne lokalne nie są obowiązkowe, więc w literale lambdy może ich w ogóle nie być. Minimalny obiekt lambda, który nie pobiera żadnych argumentów i zwraca wartość `nil`, ma następującą postać:

```
->{}
```

Jedną z zalet tej składni jest jej zwięzłość. Może to być pomocne w razie potrzeby przekazania lambdy jako argumentu do metody lub do innej lambdy:

```
def compose(f,g)          # Kombinacja dwóch lambd.  
  ->(x) { f.call(g.call(x)) }  
end  
succOfSquare = compose(->x{x+1}, ->x{x*x})  
succOfSquare.call(4)      #=> 17: Wynik działania (4*4)+1.
```

Literaly lambd tworzą obiekty klasy `Proc` i nie są tym samym co bloki. Aby do metody przyjmującej blok przekazać literal lambdy, należy przed tym literałem postawić prefiks `&`, tak jak w przypadku każdego innego obiektu klasy `Proc`. Poniżej przedstawiamy przykładowy sposób posortowania tablicy liczb w malejącej kolejności przy użyciu bloku i literala lambdy:

```
data.sort { |a,b| b-a }    # Blok.  
data.sort &->(a,b){ b-a } # Literal lambdy.
```

Jak widać, w tym przypadku prostsza jest zwykła składnia blokowa.

6.5.2. Wywoływanie obiektów proc i lambda

Obiekty proc i lambda nie są metodami i nie można wywoływać ich w taki sam sposób jak metod. Jeśli p odwołuje się do obiektu klasy Proc, nie można p wywołać jako metody. Jednak ponieważ p jest obiektem, można wywołać dowolną metodę tego obiektu. Pisaliśmy wcześniej, że klasa Proc definiuje metodę call. Wywołanie jej powoduje wykonanie kodu z oryginalnego bloku. Argumenty przekazane do tej metody stają się argumentami bloku, a jego wartość zwrotna staje się wartością zwrotną metody call:

```
f = Proc.new { |x,y| 1.0/(1.0/x + 1.0/y) }  
z = f.call(x,y)
```

Klasa Proc definiuje także operator dostępu do tablicy, który działa w taki sam sposób jak metoda call. Oznacza to, że obiekty proc i lambda można wywoływać przy użyciu takiej samej składni jak w wywołaniach metod, w których nawiasy okrągłe zastąpiono nawiasami kwadratowymi. Na przykład przedstawione powyżej wywołanie obiektu proc można zastąpić poniższym kodem:

```
z = f[x,y]
```

Ruby 1.9 pozwala na wywoływanie obiektów klasy Proc w jeszcze inny sposób. Zamiast nawiasów kwadratowych można używać nawiasów okrągłych poprzedzonych kropką:

```
z = f.(x,y)
```

Zapis .() wygląda jak wywołanie metody bez jej nazwy. Nie jest to operator, który można by było zdefiniować, a specjalny zapis składniowy wywołujący metodę call. Może być używany z każdym obiektem definiującym metodę call i nie jest ograniczony tylko do obiektów klasy Proc.

Ruby 1.9 rozszerza klasę Proc o metodę curry. Wywołanie tej metody powoduje zwrócenie przekształconej wersji obiektu proc lub lambdy. Gdy tak przekształcona wersja obiektu proc lub lambdy zostanie wywołana z niewystarczającą liczbą argumentów, zwracany jest nowy obiekt proc lub lambda (także przekształcony) z zastosowanymi argumentami, które zostały podane. Jest to często stosowana technika w programowaniu funkcyjnym:

```
product = ->(x,y){ x*y } # Definicja lambdy.  
triple = product.curry[3] # Przekształcenie, a potem zastosowanie pierwszego argumentu.  
[triple[10],triple[20]]    #=> [30,60].  
lambda {|w,x,y,z| w+x+y+z}.curry[1][2,3][4] #=> 10.
```

6.5.3. Krotność obiektów klasy Proc

Krotność (ang. *arity*) obiektów klasy Proc wyznacza liczbę wymaganych przez nie argumentów. Obiekty klasy Proc posiadają metodę o nazwie arity zwracającą liczbę wymaganych przez nie argumentów. Na przykład:

```
lambda{||}.arity      #=> 0. Nie wymaga żadnych argumentów.  
lambda{|x| x}.arity   #=> 1. Wymaga jednego argumentu.  
lambda{|x,y| x+y}.arity #=> 2. Wymaga dwóch argumentów.
```

Pojęcie krotności komplikuje się, gdy obiekt klasy Proc przyjmuje dowolną liczbę argumentów w ostatnim argumencie z prefiksem *. Kiedy obiekt klasy Proc pozwala na podawanie argumentów opcjonalnych, metoda arity zwraca liczbę ujemną wynoszącą $-n-1$. Taka wartość zwrotna oznacza, że obiekt klasy Proc wymaga n argumentów, ale może opcjonalnie pobrać jeszcze inne dodatkowe argumenty. Zapis $-n-1$ jest nazywany uzupełnieniem jedynkowym n . Można go odwrócić za pomocą operatora ~. Jeśli zatem metoda arity zwróci ujemną liczbę m , wynik działania $\sim m$ (lub $-m-1$) wyznacza liczbę wymaganych argumentów:

```
lambda {||*args|}.arity      #=> -1. ~-1 = -(1)-1 = 0 wymaganych argumentów.  
lambda {|first, *rest|}.arity #=> -2. ~-2 = -(2)-1 = 1 wymagany argument.
```

Jest jeden problem z metodą arity. W Ruby 1.8 obiekt klasy Proc zadeklarowany bez klauzuli z argumentami (to znaczy bez znaków ||) może zostać wywołany przy użyciu dowolnej liczby argumentów (argumenty te są ignorowane). Metoda arity zwraca wtedy wartość -1 , co oznacza, że nie są wymagane żadne argumenty. W Ruby 1.9 zostało to zmienione. Obiekt klasy Proc zadeklarowany w taki sposób ma krotność rzędu 0 . Jeśli jest lambdą, błędem jest wywoływanie go przy użyciu jakichkolwiek argumentów:

```
puts lambda {}.arity # -1 w 1.8; 0 w Ruby 1.9.
```

6.5.4. Porównywanie obiektów klasy Proc

W klasie Proc znajduje się metoda == służąca do sprawdzania, czy dwa obiekty tej klasy są sobie równe. Ważne jest jednak, aby zrozumieć, że do równości dwóch obiektów proc lub lambda nie wystarczy, aby miały one taki sam kod źródłowy:

```
lambda { |x| x*x } == lambda { |x| x*x } #=> false.
```

Metoda == zwraca wartość true tylko wtedy, gdy jeden obiekt klasy Proc jest klonem lub duplikatem drugiego:

```
p = lambda { |x| x*x }  
q = p.dup  
p == q          #=> true: obiekty są równe.  
p.object_id == q.object_id #=> false: nie są tym samym obiektem.
```

6.5.5. Różnica między obiektami proc i lambda

Obiekt proc jest obiektową reprezentacją bloku kodu i zachowuje się jak blok. Lambda zachowuje się nieco inaczej, bardziej jak metoda niż jak blok. Wywołanie obiektu proc przypomina przekazanie sterowania do bloku, podczas gdy wywołanie lambdy jest jak wywołanie metody. W Ruby 1.9 do sprawdzania, czy obiekt jest typu proc czy lambda, służy metoda obiektowa lambda?. Predykat ten zwraca wartość true dla lambd i false dla obiektów proc. Szczegółowe różnice pomiędzy obiektami proc a lambdami zostały opisane w poniższych podrozdziałach.

6.5.5.1. Instrukcja return w blokach, obiektach proc i lambdach

Przypomnijmy z rozdziału 5., że instrukcja return zwraca wartość z metody nadzędnej, nawet jeśli znajduje się w bloku. Instrukcja return w bloku nie powoduje powrotu tylko z bloku do wywołującego iteratora, ale powoduje wyjście z metody, która ten iterator wywołała. Na przykład:

```

def test
  puts "wejście do metody"
  1.times { puts "wejście do bloku"; return } # Zmusza do wyjścia z metody test.
  puts "wyjście z metody" # Ten wiersz nie jest nigdy wykonywany.
end
test

```

Obiekt proc jest jak blok. Zatem jeśli w takim obiekcie zostanie wykonana instrukcja `return`, nastąpi próba powrotu z metody zawierającej blok, który został przekonwertowany na obiekt `proc`. Na przykład:

```

def test
  puts "wejście do metody"
  p = Proc.new { puts "wejście do obiektu proc"; return }
  p.call # Wywołanie obiektu proc zmusza metodę do powrotu.
  puts "wyjście z metody" # Ten wiersz nie jest nigdy wykonywany.
end
test

```

Jednak używając instrukcji `return` w obiektach `proc`, można wpasować w pułapkę, ponieważ obiekty te są często przekazywane pomiędzy różnymi metodami. Zanim obiekt `proc` zostanie wywołany, metoda nadzędna może być już po zwrocie wartości:

```

def procBuilder(message) # Utworzenie i zwrócenie obiektu proc.
  Proc.new { puts message; return } # Instrukcja return powoduje wyjście z metody procBuilder.
  # Ale metoda procBuilder zwróciła już wartość tutaj!
end
def test
  puts "wejście do metody"
  p = procBuilder("wejście do obiektu proc")
  p.call # Drukuje "wejście do obiektu proc" i zgłasza wyjątek LocalJumpError!
  puts "wyjście z metody" # Ten wiersz nie jest nigdy wykonywany.
end
test

```

Dzięki konwersji bloku na obiekt możliwe było przekazanie go między różnymi metodami i użycie całkiem poza kontekstem. W takiej sytuacji istnieje ryzyko, że nastąpi próba zmuszenia do zwrócenia wartości z metody, która już tę wartość zwróciła — jak w powyższym przypadku. Wtedy Ruby zgłasza wyjątek `LocalJumpError`.

Rozwiązaniem tego naciąganego problemu jest oczywiście usunięcie niepotrzebnej instrukcji `return`. Niestety, nie zawsze instrukcja `return` jest zbędna. Wtedy rozwiązaniem jest użycie lambdy zamiast obiektu `proc`. Jak napisaliśmy wcześniej, lambdy są bardziej podobne do metod niż do bloków. Zatem instrukcja `return` w lambdzie powoduje zwrot wartości przez samą lambdę, a nie przez metodę, w której ta lambda została utworzona:

```

def test
  puts "wejście do metody"
  p = lambda { puts "wejście do lambdy"; return }
  p.call # Wywołanie lambdy nie zmusza metody do powrotu.
  puts "wyjście z metody" # Ten wiersz *jest* tym razem wykonywany.
end
test

```

Dzięki temu że instrukcja `return` w lambdach powoduje tylko zwrot wartości przez same lambdy, nie trzeba przejmować się wyjątkiem `LocalJumpError`:

```

def lambdaBuilder(message) # Utworzenie i zwrócenie lambdy.
  lambda { puts message; return } # Instrukcja return zmusza lambdę do zwrotu wartości.
end
def test
  puts "wejście do metody"

```

```

l = lambdaBuilder("wejście do lambdy")
l.call          # Drukuje "wejście do lambdy".
puts "wyjście z metody" # Ten wiersz jest wykonywany.
end
test

```

6.5.5.2. Instrukcja break w blokach, obiektach proc i lambdach

Rysunek 5.3 ilustruje działanie instrukcji `break` w bloku. Zmusza ona blok do zwrócenia wartości do swojego iteratora, a ten do zwrócenia wartości do metody, która go wywołała. Ponieważ obiekty `proc` są podobne do bloków, można się spodziewać się, że instrukcja `break` w nich działa podobnie. Nie da się jednak tego łatwo sprawdzić. Kiedy tworzony jest obiekt `proc` za pomocą metody `Proc.new`, metoda `Proc.new` jest iteratorem, z którego zwrażałaby instrukcja `break`. Zanim będzie można wywołać ten obiekt `proc`, iterator zwróci już wartość. Dlatego nie ma sensu używanie instrukcji `break` na najwyższym poziomie w obiektach `proc` utworzonych za pomocą metody `Proc.new`:

```

def test
  puts "wejście do metody test"
  proc = Proc.new { puts "wejście do obiektu proc"; break }
  proc.call          # LocalJumpError: iterator już zwrócił wartość.
  puts "wyjście z metody test"
end
test

```

Jeśli obiekt `proc` zostanie utworzony z argumentem & w metodzie iteracyjnej, można go wywołać i zmusić iterator do powrotu:

```

def iterator(&proc)
  puts "wejście do iteratora"
  proc.call          # Wywołanie obiektu proc.
  puts "wyjście z iteratora"    # Kod ten nie jest wykonywany, jeśli obiekt proc wywołuje instrukcję break.
end
def test
  iterator { puts "wejście do obiektu proc"; break }
end
test

```

Lambdy przypominają metody, więc umieszczenie instrukcji `break` na samej górze lambdy, bez otaczającej pętli lub iteracji, z której można by było wyjść, nie ma faktycznie sensu! Można się spodziewać, że poniższy kod spowoduje wyjątek, ponieważ w lambdzie nie ma nic, czego działanie można by było przerwać. W rzeczywistości instrukcja `break` umieszczona na najwyższym poziomie działa jak instrukcja `return`:

```

def test
  puts "wejście do metody test"
  lambda = lambda { puts "wejście do lambdy"; break; puts "wyjście z lambdy" }
  lambda.call
  puts "wyjście z metody test"
end
test

```

6.5.5.3. Inne instrukcje kontrolujące przepływ sterowania

Instrukcja `next` znajdująca się na najwyższym poziomie w blokach, obiektach `proc` i lambdach działa tak samo — zmusza instrukcję `yield` lub metodę `call`, która wywołała ten blok, obiekt `proc` lub lambdę do zwrotu wartości. Jeśli po instrukcji `next` znajduje się jakieś wyrażenie, jego wartość jest wartością zwrotną bloku, obiektu `proc` lub lambdy.

Słowo kluczowe `redo` również działa tak samo w obiektach `proc` i `lambdach` — przekazuje sterowanie z powrotem na ich początek.

Słowa kluczowego `retry` w `lambdach` i obiektach `proc` nie można używać — jego użycie zawsze powoduje wyjątek `LocalJumpError`.

Metoda `raise` działa tak samo w blokach, obiektach `proc` i `lambdach`. Wyjątki zawsze są propagowane w górę stosu wywołań. Jeśli blok, obiekt `proc` lub `lambda` spowoduje wyjątek i nie ma żadnej lokalnej klawzuli `rescue` do jego obsługi, wyjątek ten jest najpierw przesyłany do metody, która wywołała dany blok za pomocą instrukcji `yield` lub wywołała obiekt `proc` lub `lambda` za pomocą metody `call`.

6.5.5.4. Przekazywanie argumentów do obiektów `proc` i `lambd`

Wywołanie bloku za pomocą instrukcji `yield` jest podobne, ale nie identyczne z wywołaniem metody. Różnice dotyczą sposobu przypisywania wartości argumentów wywołania do zmiennych argumentowych zadeklarowanych w bloku lub metodzie. Instrukcja `yield` wykorzystuje **semantykę `yield`**, a w wywołaniu metody jest wykorzystywana **semantyka wywołań**. Semantyka instrukcji `yield` jest podobna do przypisania równoległego i została opisana w podrozdziale 5.4.4. Jak się można spodziewać, w wywołaniu obiektu `proc` wykorzystywana jest semantyka instrukcji `yield`, a w wywołaniu `lambda` semantyka wywołań:

```
p = Proc.new { |x,y| print x,y }
p.call(1)      #x,y=1: w miejscu brakującej r-wartości zostanie użta wartość nil: Drukuje Inil.
p.call(1,2)    #x,y=1,2: 2 l-wartości, 2 r-wartości:   drukuje 12.
p.call(1,2,3)  #x,y=1,2,3: dodatkowa r-wartość zostaje odrzucona: drukuje 12.
p.call([1,2])  #x,y=[1,2]: tablica zostaje automatycznie rozpakowana: drukuje 12.
```

Niniejszy kod demonstruje, że metoda `call` obiektu `proc` elastycznie obsługuje odbierane argumenty — po cichu odrzuca nadwyżki, dodaje wartości `nil` dla opuszczonych argumentów, a nawet rozpakowuje tablice (oraz, co nie zostało tutaj pokazane, pakuje argumenty w jedną tablicę, kiedy obiekt `proc` wymaga tylko jednego argumentu).

Lambdy nie są tak elastyczne. Podobnie jak w przypadku metod, w ich wywołaniu musi zostać podana dokładnie taka sama liczba argumentów jak w deklaracji:

```
l = lambda { |x,y| print x,y }
l.call(1,2)      #To działa.
l.call(1)        #Zla liczba argumentów.
l.call(1,2,3)    #Zla liczba argumentów.
l.call([1,2])    #Zla liczba argumentów.
l.call(*[1,2])   #Dziala: operator splat rozpakowuje tablice.
```

6.6. Domknięcia

Obiekty `proc` i `lambda` są **domknięciami** (ang. *closure*). Termin „domknięcie” powstał na początku istnienia informatyki. Oznacza obiekt, który jest jednocześnie dającą się wywoływać funkcją i obiektem wiążącym zmienne tej funkcji. Kiedy tworzony jest obiekt `proc` lub `lambda`, powstaje obiekt klasy `Proc` zawierający nie tylko wykonywalny blok, ale również wiązania do wszystkich zmiennych użytych w tym bloku.

Wiadomo już, że bloki mogą używać zmiennych i argumentów metod, które zostały zdefiniowane poza nimi. Na przykład w poniższym fragmencie kodu blok związany z iteratorem `collect` wykorzystuje argument `n` metody `multiply`:

```
# Mnoży każdy element tablicy data przez n.
def multiply(data, n)
  data.collect { |x| x*n }
end
puts multiply([1,2,3], 2)  # Drukuje 2,4,6.
```

Jeszcze ciekawsze i bardziej zaskakujące jest to, że jeśli blok ten zostałby zamieniony w obiekt proc lub lambdę, miałby dostęp do `n` nawet po tym, jak metoda, której `n` jest argumentem, zwróciłaby już wartość. Na przykład:

```
# Zwraca lambdę, która zachowuje argument n.
def multiplier(n)
  lambda { |data| data.collect{ |x| x*n } }
end
doubler = multiplier(2)      # Użycie lambdy, która podwaja każdą wartość.
puts doubler.call([1,2,3])  # Drukuje 2,4,6.
```

Metoda `multiplier` zwraca lambdę. Ponieważ lambda ta została użyta poza zakresem, w którym jest zdefiniowana, nosi nazwę domknięcia. Zawiera w sobie lub inaczej „domyka” (lub zachowuje) wiązanie do argumentu metody `n`.

6.6.1. Domknięcia i współdzielone zmienne

Ważne jest, aby pamiętać, że domknięcie nie tylko zachowuje wartości zmiennych, do których się odwołuje, ale zachowuje same zmienne, przedłużając tym samym ich cykl życia. Innymi słowy, zmienne używane w obiektach proc lub lambdach nie są statycznie związane podczas tworzenia tych obiektów. Ich wiązania są dynamiczne, a więc wartości zmiennych są odszuukiwane w chwili wykonywania lambdy lub obiektu proc.

Na przykład w poniższym programie zdefiniowano metodę zwracającą dwie lambdy. Ponieważ oba te obiekty są zdefiniowane w tym samym zakresie, mają dostęp do tych samych jego zmiennych. Kiedy jedna lambda zmieni wartość którejś ze wspólnych zmiennych, zmiana ta jest widoczna w drugiej lambdzie:

```
# Zwraca dwie lambdy mające dostęp do tej samej zmiennej lokalnej.
def accessor_pair(initialValue=nil)
  value = initialValue # Zmienna lokalna wspólnie utworzone lambdy.
  getter = lambda { value } # Zwraca wartości zmiennej lokalnej.
  setter = lambda { |x| value = x } # Zmiana wartości zmiennej lokalnej.
  return getter, setter # Zwraca dwie lambdy do algorytmu wywołującego.
end
getX, setX = accessor_pair() # Utworzenie lambd dostępowych dla początkowej wartości 0.
puts getX[]          # Drukuje 0. Zamiast metody call użyto nawiasów kwadratowych.
setX[10]           # Zmiana wartości przez jedno z domknięć.
puts getX[]          # Drukuje 10. Zmiana jest widoczna w drugim domknięciu.
```

Dostęp lambd utworzonych w tym samym zakresie do tych samych zmiennych może być zaletą, ale też źródłem błędów. Za każdym razem, gdy jakaś metoda zwraca więcej niż jedno domknięcie, należy zwrócić szczególną uwagę na to, do jakich zmiennych domknięcie te mają dostęp. Przeanalizuj poniższy fragment kodu:

```
# Zwraca tablicę lambd mnożonych przez argumenty.
def multipliers(*args)
  x = nil
  args.map { |x| lambda { |y| x*y } }
end
double, triple = multipliers(2,3)
puts double.call(2)    # Drukuje 6 w Ruby 1.8.
```

Niniejsza metoda `multipliers` wykorzystuje iterator `map` i związany z nim blok kodu do zwrócenia tablicy lambd (utworzonych wewnątrz tego bloku). W Ruby 1.8 argumenty blokowe nie zawsze mają zasięg ograniczony do swoich bloków (zobacz podrozdział 5.4.3), w wyniku czego wszystkie lambdy mają dostęp do `x` — zmiennej lokalnej metody `multipliers`. Jak pamiętasz, domknięcia nie przechwytują bieżącej wartości zmiennej, tylko całą zmienną. Każda z utworzonych tu lambd dysponuje zmienną `x`. Ma ona tylko jedną wartość wykorzystywaną przez wszystkie mające do niej dostęp lambdy. Dlatego właśnie lambda, którą nazwaliśmy `double` (podwójny), potroiła swój argument, zamiast go podwoić.

Problem ten w tym konkretnym przypadku nie dotyczy Ruby 1.9, ponieważ w tej wersji języka argumenty blokowe mają zawsze zasięg lokalny ograniczony do ich bloku. Nadal jednak można mieć problemy, jeśli utworzy się lambdę w pętli i w tej lambdzie umieści zmienną pętlową (na przykład indeks tablicy).

6.6.2. Domknięcia i wiązania

W klasie `Proc` dostępna jest metoda o nazwie `binding`. Wywołanie jej na rzecz obiektu `proc` lub lambdy powoduje zwrot obiektu klasy `Binding` reprezentującego efektywne wiązania w danym domknięciu.

Więcej o wiązaniach

Opisujemy wiązania domknięć w taki sposób, jakby były zwykłymi odwzorowaniemi nazw zmiennych na wartości zmiennych. W rzeczywistości jednak wiązania to coś więcej niż tylko zmienne. Zawierają one pełne informacje potrzebne do wykonania metody, jak wartość obiektu `self` i blok, jeśli istnieje, który został wywołany przez instrukcję `yield`.

Obiekt klasy `Binding` sam w sobie nie udostępnia żadnych ciekawych metod, ale może zostać użyty jako drugi argument globalnej funkcji o nazwie `eval` (zobacz podrozdział 8.2), dostarczając kontekst do obliczenia łańcucha kodu Ruby. W Ruby 1.9 klasa `Binding` posiada własną metodę `eval` (więcej na temat metod `Kernel.eval` i `Binding.eval` można znaleźć za pomocą narzędzia `ri`).

Dzięki obiektom klasy `Binding` i metodzie `eval` programista ma możliwość skorzystania z tego dojścia pozwalającego na manipulację zachowaniem domknięcia. Spójrz jeszcze raz na prezentowany wcześniej fragment kodu:

```
# Zwraca lambdę, która zachowuje argument n.
def multiplier(n)
  lambda {|data| data.collect{|x| x*n} }
end
doubler = multiplier(2)      # Użycie lambdy, która podwaja każdą wartość.
puts doubler.call([1,2,3])   # Drukuje 2,4,6.
```

Założmy, że chcesz zmienić zachowanie domknięcia `doubler`:

```
eval("n=3", doubler.binding) # Lub doubler.binding.eval("n=3") w Ruby 1.9.
puts doubler.call([1,2,3])   # Teraz drukuje 3,6,9!
```

Metoda `eval` pozwala na przekazanie obiektu klasy `Proc` bezpośrednio, zamiast przekazywać obiekt klasy `Binding` tego obiektu klasy `Proc`. W związku z tym wywołanie metody `eval` w powyższym kodzie można zastąpić następującym:

```
eval("n=3", doubler)
```

Wiązania nie są wyłączną właściwością domknięć. Metoda `Kernel.binding` zwraca obiekty klasy `Binding` reprezentujące efektywne wiązania w dowolnym miejscu, w którym została wywołana.

6.7. Obiekty klasy Method

Metody i bloki są w języku Ruby konstrukcjami, które można wywoływać, ale nie są obiektami. Obiekty `proc` i `lambda` są obiektowymi wersjami bloków. Można je wywoływać i wykonywać na nich operacje jak na danych. W języku Ruby dostępny jest bardzo rozbudowany mechanizm metaprogramowania (czyli **refleksji**), a metody mogą być reprezentowane jako egzemplarze klasy `Method` (metaprogramowanie opisane jest w rozdziale 8., natomiast obiekty klasy `Method` zostały przedstawione tutaj). Należy pamiętać, że wywołanie metody poprzez obiekt klasy `Method` jest wolniejsze niż wywołanie bezpośrednie. Obiekty klasy `Method` są rzadziej używane od obiektów `proc` i `lambda`.

W klasie `Object` dostępna jest metoda o nazwie `method`. Jeśli zostanie do niej przekazana nazwa metody w postaci łańcucha lub symbolu, zwróci ona obiekt klasy `Method` reprezentujący tę metodę klasy swojego adresata, której nazwa została przekazana (lub zgłosi wyjątek `NameError`, jeśli taka metoda nie istnieje). Na przykład:

```
m = 0.method(:succ) # Obiekt klasy Method reprezentujący metodę succ obiektu 0 klasy Fixnum.
```

W Ruby 1.9 obiekty klasy `Method` można także tworzyć za pomocą metody `public_method`. Działa ona podobnie jak metoda `method`, ale ignoruje metody prywatne i chronione (zobacz podrozdział 7.2).

Klasa `Method` nie jest podklassą klasy `Proc`, ale swoim zachowaniem w dużym stopniu tę klasę przypomina. Obiekty klasy `Method` wywołuje się za pomocą metody `call` (lub operatora `[]`), tak samo jak obiekty klasy `Proc`. Ponadto klasa `Method`, podobnie jak `Proc`, udostępnia metodę `arity`. Wywołanie obiektu `m` klasy `Method`:

```
puts m.call # To samo co puts 0.succ. lub puts m[].
```

Wywołanie metody przez obiekt klasy `Method` nie wymaga zmian w semantyce wywołania ani nie zmienia działania instrukcji sterujących przepływem jak `return` i `break`. Metoda `call` obiektów klasy `Method` wymaga stosowania semantyki wywoływanego metod, a nie instrukcji `yield`. W związku z tym obiekty klasy `Method` bardziej przypominają lambdy niż obiekty `proc`.

Obiekty klasy `Method` działaniem bardzo przypominają obiekty klasy `Proc` i z reguły można je stosować zamiennie. Kiedy wymagany jest prawdziwy obiekt klasy `Proc`, można obiekt klasy `Method` odpowiednio przekonwertować za pomocą metody `Method.to_proc`. Dlatego właśnie przed obiektami klasy `Method` można stawiać prefiks `&` i przesyłać je do metod zamiast bloków. Na przykład:

```
def square(x); x*x; end
puts (1..10).map(&method(:square))
```

Jedną ważną różnicą pomiędzy obiektami klasy `Method` a obiektami klasy `Proc` jest to, że te pierwsze nie są domknięciami. Metody w języku Ruby powinny być w pełni samowystarczalne, przez co nie mają dostępu do zmiennych lokalnych spoza swojego zakresu. W związku z tym jedyne wiązanie zachowywane przez obiekty klasy `Method` to wartość obiektu `self` — obiektu, na rzecz którego metoda ma zostać wywołana.

Definiowanie metod za pomocą obiektów klasy Proc

Poza tworzeniem obiektów klasy Method reprezentujących metody i konwertowaniem ich na obiekty klasy Proc można także dokonywać operacji odwrotnej. Metoda `define_method` (klasy `Module`) przyjmuje jako argument obiekt klasy `Symbol` i tworzy metodę o takiej nazwie, której całem jest związany z nią blok kodu. Zamiast bloku można także do tej metody przekazać obiekt klasy Proc lub Method na miejscu drugiego argumentu.

W Ruby 1.9 klasa `Method` udostępnia trzy metody, których nie ma w Ruby 1.8: `name` zwraca nazwę metody w postaci łańcucha; `owner` zwraca nazwę klasy, w której została zdefiniowana; a `receiver` zwraca obiekt, z którym jest związana. Dla każdego obiektu `m`, `m.receiver.class` musi być równy lub być podklassą `m.owner`.

6.7.1. Niezwiązane obiekty klasy Method

Poza klasą `Method` w języku Ruby dostępna jest też klasa `UnboundMethod`. Jak wskazuje jej nazwa, obiekty klasy `UnboundMethod` reprezentują metody nieposiadające dowiązań do obiektów, na rzecz których mają zostać wywołane. Ponieważ obiekt klasy `UnboundMethod` nie jest z niczym powiązany, nie można go wywołać, w związku z czym klasa `UnboundMethod` nie udostępnia metody `call` ani `[]`.

Do tworzenia obiektów klasy `UnboundMethod` służy metoda `instance_method` dowolnej klasy lub dowolnego modułu:

```
unbound_plus = Fixnum.instance_method("+")
```

W Ruby 1.9 obiekty klasy `UnboundMethod` można także tworzyć za pomocą metody `public_>instance_method`. Działa ona tak samo jak metoda `instance_method`, z tym, że ignoruje metody prywatne i chronione (zobacz podrozdział 7.2).

Aby wywołać niezwiązaną metodę trzeba ją najpierw związać z jakimś obiektem za pomocą metody `bind`:

```
plus_2 = unbound_plus.bind(2) # Zwiążanie metody z obiektem 2.
```

Metoda `bind` zwraca obiekt klasy `Method`, który można wywołać za pomocą metody `call`:

```
sum = plus_2.call(2) # => 4.
```

Innym sposobem na utworzenie obiektu klasy `UnboundMethod` jest użycie metody `unbind` z klasą `Method`:

```
plus_3 = plus_2.unbind.bind(3)
```

W Ruby 1.9 klasa `UnboundMethod` udostępnia metody `name` i `owner` działające tak samo jak w klasie `Method`.

6.8. Programowanie funkcyjne

Język Ruby nie jest językiem funkcyjnym w takim samym sensie jak Lisp i Haskell, ale jego bloki oraz obiekty `proc` i `lambda` bardzo dobrze nadają się do tego stylu programowania. Używając bloku z iteratorem `Enumerable`, jak `map` czy `inject`, programuje się w stylu funkcyjnym.

nym. Poniżej znajdują się przykłady takiego stylu programowania z użyciem iteratorów map i inject:

```
# Oblicza średnią i standardowe odchylenie w tablicy liczb.  
mean = a.inject{|x,y| x+y } / a.size  
sumOfSquares = a.map{|x| (x-mean)**2 }.inject{|x,y| x+y }  
standardDeviation = Math.sqrt(sumOfSquares/(a.size-1))
```

Jeśli ktoś lubi programowanie funkcyjne, może z łatwością do wbudowanych klas języka Ruby dodać własności umożliwiające stosowanie tego paradygmatu. Reszta niniejszego rozdziału opisuje niektóre możliwości pracy z funkcjami. Kod prezentowany w niniejszym podrozdziale jest bardzo zwięzły, a jego celem jest poszerzenie horyzontów, a nie podanie recepty na dobry styl programowania. W szczególności przeddefiniowywanie operatorów na taką skalę jak w przykładach w poniższych podrozdziałach powoduje, że kod programów staje się bardzo trudny do odczytu i konserwacji dla innych osób!

Prezentowany tutaj materiał jest zaawansowany i został napisany przy założeniu, że Czytelnik posiada wiadomości przedstawione w rozdziale 7. Dlatego przy pierwszym czytaniu tej książki można resztę tego rozdziału pominąć.

6.8.1. Zastosowanie funkcji do obiektów umożliwiających iterację

Metody map i inject to dwa najważniejsze iteratory dostępne w module Enumerable. Każdy z nich wymaga podania bloku kodu. Pisząc programy w stylu funkcyjnym, można potrzebować metod opartych na funkcjach, które pozwalają na zastosowanie ich do wyznaczonych obiektów umożliwiających iterację:

```
# Niniejszy moduł zawiera definicje metod i operatorów przeznaczonych do programowania funkcyjnego.  
module Functional  
  # Funkcja ta zostanie zastosowana do każdego elementu wyznaczonego obiektu umożliwiającego iterację  
  # i zwróci tablicę wyników. Jest to odwrotność metody Enumerable.map.  
  # Znak | jest używany jako alias operatora. Należy go czytać „zastosowany na rzecz”.  
  #  
  # Przykład:  
  # a = [[1,2],[3,4]]  
  # sum = lambda {|x,y| x+y}  
  # sums = sum|a #=> [3,7]  
  def apply(enum)  
    enum.map &self  
  end  
  alias | apply  
  # Funkcja ta redukuje obiekt umożliwiający iterację do pojedynczej wartości.  
  # Odwrotność metody Enumerable.inject.  
  # Aliasem operatora jest <=.  
  # Wskazówka: <= wygląda jak igła do robienia zastrzyków.  
  # Przykład:  
  # data = [1,2,3,4]  
  # sum = lambda {|x,y| x+y}  
  # total = sum<=data #=> 10  
  def reduce(enum)  
    enum.inject &self  
  end  
  alias <= reduce  
end  
# Dodanie tych metod programowania funkcyjnego do klas Proc i Method.  
class Proc; include Functional; end  
class Method; include Functional; end
```

Należy zauważyc, że powyższe metody zostały zdefiniowane w module o nazwie `Functional`, który następnie został dołączony do klas `Proc` i `Method`. Dzięki temu metody `apply` i `reduce` działają zarówno na obiektach `proc`, jak i na obiektach klasy `Method`. Większość prezentowanych dalej metod również definiuje metody w module `Functional`, dzięki czemu działają one zarówno na obiektach klasy `Proc`, jak i `Method`.

Majac zdefiniowane metody `apply` i `reduce`, možesz zmodyfikowaç swoje obliczenia statystyczne w następujący sposób:

```
sum = lambda {|x| x+y }          # Funkcja dodajaca dwie liczby.  
mean = (sum<=a)/a.size           # Albo sum.reduce(a), albo a.inject(&sum).  
deviation = lambda {|x| x-mean } # Funkcja obliczajaca roznice ze sredniaj.  
square = lambda {|x| x*x }        # Funkcja podnoszaca liczbę do kwadratu.  
standardDeviation = Math.sqrt((sum<=square|(deviation|a))/(a.size-1))
```

Zauważ, że ostatni wiersz kodu jest bardzo zwięzły, ale użyte w nim niestandardowe operatory znacznie utrudniają jego czytanie. Dodatkowo istotne jest, że operator `|` wiąże lewostronnie, nawet jeśli zostanie zdefiniowany przez programistę. W związku z tym w składni stosowania kilku funkcji do obiektów umożliwiających iterację konieczne jest użycie nawiasów. To znaczy że zamiast `square|deviation|a` trzeba napisać `square|(deviation|a)`.

6.8.2. Łączenie funkcji

Majac dwie funkcje `f` i `g`, czasami konieczne jest zdefiniowanie nowej funkcji `f(g())`, czyli `f połączona z g`. Można napisać metodę, która automatycznie wykonuje ich łączenie:

```
module Functional  
  # Zwraca nową lambda obliczającą self[[f[args]]].  
  # Użycie operatora * jako alias dla metody compose.  
  # Przykłady z użyciem aliasa * dla tej metody.  
  #  
  # f = lambda {|x| x*x }  
  # g = lambda {|x| x+1 }  
  # (f*g)[2] # => 9  
  # (g*f)[2] # => 5  
  #  
  # def polar(x,y)  
  # [Math.hypot(y,x), Math.atan2(y,x)]  
  # end  
  # def cartesian(magnitude, angle)  
  # [magnitude*Math.cos(angle), magnitude*Math.sin(angle)]  
  # end  
  # p,c = method :polar, method :cartesian  
  # (c*p)[3,4] # => [3,4]  
  #  
  def compose(f)  
    if self.respond_to?(:arity) && self.arity == 1  
      lambda { |*args| self[f[*args]] }  
    else  
      lambda { |*args| self[*f[*args]] }  
    end  
  end  
  # * jest naturalnym operatorem łączenia funkcji.  
  alias * compose  
end
```

Przykładowy kod znajdujący się w komentarzach demonstruje użycie metody `compose` z obiektami klasy `Method` oraz z lambdami. Dzięki użyciu nowego operatora łączenia funkcji `*` można nieco uprościć obliczenia odchylenia standardowego. Przy użyciu tych samych definicji lambd `sum`, `square` i `deviation` kod wygląda następująco:

```
standardDeviation = Math.sqrt((sum<=square*deviation|a)/(a.size-1))
```

Różnica polega na tym, że funkcje square i deviation zostały połączone w jedną przed zastosowaniem do tablicy a.

6.8.3. Częściowa aplikacja funkcji

W programowaniu funkcyjnym mianem **częściowej aplikacji** (ang. *partial application*) określa się proces tworzenia nowej funkcji z funkcji już istniejącej i części jej argumentów. Ta nowa funkcja jest równoważna oryginalnej funkcji z ustalonymi wyznaczonymi argumentami. Jest to nieco podobne (ale nie do końca) do techniki „przyprawiania” za pomocą metody Proc.curry. Na przykład:

```
product = lambda {|x, y| x*y }          # Funkcja z dwoma argumentami.  
double = lambda {|x| product(2,x) }      # Aplikacja jednego argumentu.
```

Częściową aplikację można uprościć za pomocą odpowiednich metod (i operatorów) dodanych do modułu Functional:

```
module Functional  
  #  
  # Zwraca lambdę odpowiadającą tej z zastosowanym jednym lub większą liczbą  
  # początkowych argumentów. Kiedy podany jest tylko jeden argument,  
  # prostszy w użyciu może być alias >>.   
  # Przykład:  
  # product = lambda {|x,y| x*y}  
  # doubler = lambda >> 2  
  #  
  def apply_head(*first)  
    lambda { |*rest| self[*first.concat(rest)]}  
  end  
  #  
  # Zwraca lambdę odpowiadającą tej z zastosowanym jednym lub większą liczbą końcowych  
  # argumentów. Kiedy podany jest tylko jeden argument,  
  # prostszy może być alias <<.   
  # Przykład:  
  # difference = lambda {|x,y| x-y}  
  # decrement = difference << 1  
  #  
  def apply_tail(*last)  
    lambda { |*rest| self[*rest.concat(last)]}  
  end  
  # Alternatywne operatory dla tych metod. Nawiasy ostre  
  # wskazują, po której stronie argument jest wsuwany.  
  alias >> apply_head    # g = f>> 2 -- ustawienie pierwszego argumentu na 2.  
  alias << apply_tail    # g = f<< 2 -- ustawienie ostatniego argumentu na 2.  
end
```

Przy użyciu tych metod i operatorów możliwe jest zdefiniowanie funkcji double po prostu jako product>>2. Za pomocą częściowej aplikacji można sprawić, że obliczenia odchylenia standardowego będą nieco bardziej abstrakcyjne. Funkcję deviation zbuduj na podstawie ogólniej-szej funkcji difference:

```
difference = lambda {|x,y| x-y }  # Oblicza różnicę dwóch liczb.  
deviation = difference<<mean      # Aplikacja drugiego argumentu.
```

6.8.4. Spamiętywanie funkcji

Terminem **spamiętywanie** (ang. *memoization*) w programowaniu funkcyjnym określa się czynność zapisywania w pamięci wyników wywołania funkcji. Jeśli funkcja zawsze zwraca tę samą wartość po podaniu tych samych argumentów, są podstawy, aby uznać, że te same argumenty będą używane wielokrotnie. W przypadku gdy obliczenia wykonywane przez funkcję są czasochłonne, spamiętywanie może znacznie przyspieszyć działanie programu. Spamiętywanie można zautomatyzować dla obiektów klas Proc i Method za pomocą następującej metody:

```
module Functional
  #
  # Zwraca nową lambdę, która zapamiętuje wyniki tej funkcji i
  # wywołuje ją tylko wówczas, gdy zostaną podane nowe argumenty.
  #
  def memoize
    cache = {} # Pusta pamięć podręczna. Lambda obejmuje ją w swoim domknięciu.
    lambda { |*args|
      # Zauważ, że klucz tablicy asocjacyjnej jest całą tablicą argumentów!
      unless cache.has_key?(args) # Jeśli nie ma jeszcze zapisanych wyników dla tych argumentów,
        cache[args] = self[*args] # wykonuje obliczenia i zapisuje wynik.
      end
      cache[args] # Zwraca wynik z pamięci podręcznej.
    }
  end
  # Jednoargumentowy operator + (prawdopodobnie niepotrzebny) dla spamiętywania.
  # Wskazówka: operator + oznacza "ulepszony".
  alias +@ memoize # cached_f = +f
end
```

Poniżej przedstawiony jest przykładowy sposób użycia metody `memoize` lub jednoargumentowego operatora `+`:

```
# Spamiętywana rekursywna funkcja factorial.
factorial = lambda { |x| return 1 if x==0; x*factorial[x-1]; }.memoize
# Użycie operatora +.
factorial = +lambda { |x| return 1 if x==0; x*factorial[x-1]; }
```

Warto zwrócić uwagę, że prezentowana tu funkcja `factorial` jest rekursywna. Wywołuje spamiętaną wersję samej siebie, co umożliwia maksymalną optymalizację. Nie działałaby tak dobrze, gdyby została zdefiniowana rekursywna niespamiętywana wersja tej funkcji, a następnie zdefiniowano by jej spamiętywaną wersję:

```
factorial = lambda { |x| return 1 if x==0; x*factorial[x-1]; }
cached_factorial = +factorial # Wywołania rekursywne nie są zapisywane w pamięci podręcznej!
```

6.8.5. Klasy Symbol, Method i Proc

Klasy `Symbol`, `Method` i `Proc` są ze sobą blisko spokrewnione. Znasz już metodę o nazwie `method`, która przyjmuje jako argument obiekt klasy `Symbol` i zwraca obiekt klasy `Method`.

W Ruby 1.9 klasa `Symbol` zyskała bardzo przydatną metodę `to_proc`. Metoda ta pozwala na postawienie przed symbolem prefiksu & i przekazanie go jako bloku do iteratatora. Symbol ten jest traktowany jako nazwa metody. Kiedy obiekt klasy `Proc` utworzony za pomocą metody `to_proc` zostaje wywołany, uruchamia metodę swojego pierwszego argumentu o nazwie wyznaczonej przez ten symbol. Pozostałe argumenty są przekazywane do tej wyznaczonej metody. Oto przykład zastosowania opisanej własności:

```
# Zwiększenie tablicy liczb całkowitych za pomocą metody Fixnum.succ.  
[1,2,3].map(&:succ) # => [2,3,4].
```

Gdyby nie metoda `Symbol.to_proc`, kod musiałby być nieco bardziej rozwlekły:

```
[1,2,3].map { |n| n.succ }
```

Początkowo metoda `Symbol.to_proc` miała być rozszerzeniem języka Ruby 1.8 i zazwyczaj jest implementowana w następujący sposób:

```
class Symbol  
  def to_proc  
    lambda {|receiver, *args| receiver.send(self, *args)}  
  end  
end
```

W tej implementacji do wywołania metody wyznaczonej przez symbol użyto metody `send` (zobacz podrozdział 8.4.3). Implementacja ta mogłaby również wyglądać następująco:

```
class Symbol  
  def to_proc  
    lambda {|receiver, *args| receiver.method(self)[*args]}  
  end  
end
```

Poza metodą `to_proc` można zdefiniować kilka innych podobnych i czasami bardzo przydatnych narzędzi. Zacznijmy od klasy `Module`:

```
class Module  
  # Dostęp do metod obiektowych przy użyciu notacji tablicowej. Zwraca obiekt klasy UnboundMethod.  
  alias [] instance_method  
end
```

W tym przypadku zdefiniowany został skrót do metody `instance_method` klasy `Module`. Przypomnijmy, że metoda ta zwraca obiekty klasy `UnboundMethod`, których nie można wywoływać, dopóki nie zostaną zвязane z konkretnym obiektem swojej klasy. Poniżej przedstawiony jest przykład użycia tej nowej notacji (zauważ urok indeksowania klasy nazwami jej metod!):

```
String[:reverse].bind("hello").call #=> "olleh".
```

Wiązanie niezwiązań z niczym metod można również uprościć przy użyciu niewielkiej ilości tego samego cukru syntaktycznego:

```
class UnboundMethod  
  # Zezwolenie na używanie [] jako alternatywnego sposobu wiązania.  
  alias [] bind  
end
```

Dzięki temu aliasowi i przy użyciu istniejącego aliasu `[]` do wywoływania metod niniejszy kod może wyglądać następująco:

```
String[:reverse]["hello"][] #=> "olleh".
```

Pierwsza para nawiasów kwadratowych indeksuje metodę, druga ją wiąże, a trzecia wywołuje.

Następnie jeśli operator `[]` ma być używany do wyszukiwania metod obiektowych klas, to operatora `[]=` można użyć do definiowania metod obiektowych:

```
class Module  
  # Definicja metody obiektowej o nazwie sym i ciele f.  
  # Przykład: String[:backwards] = lambda { reverse }  
  def []=(sym, f)  
    self.instance_eval { define_method(sym, f) }  
  end  
end
```

Definicja tego operatora `[]=` może wydawać się zawiła — jest to zaawansowane programowanie w języku Ruby. Metoda `define_method` jest prywatną metodą klasy `Module`. Metoda `instance_eval` (publiczna metoda klasy `Object`) służy do uruchamiania bloków (także wywoływanie metod prywatnych), tak jakby były one wewnątrz modułu, w którym metoda ta jest zdefiniowana. Metody `instance_eval` i `define_method` opisane są jeszcze raz w rozdziale 8.

Użyj nowego operatora `[]=` do zdefiniowania metody `Enumerable.average`:

```
Enumerable[:average] = lambda do
  sum, n = 0.0, 0
  self.each { |x| sum += x; n += 1 }
  if n == 0
    nil
  else
    sum/n
  end
end
```

W tym przypadku użyte zostały operatory `[]` i `[]=` do uzyskania i ustawienia metod obiektowych klasy lub modułu. Coś podobnego można zrobić dla metod singletonowych obiektu (do których zaliczają się metody klasowe klas i modułów). Każdy obiekt może posiadać metodę singletonową, ale nie ma sensu definiować operatora `[]` w klasie `Object`, ponieważ bardzo dużo podklas tej klasy już zawiera jego definicję. W związku z tym dla metod singletonowych można przyjąć odwrotny kurs działania i zdefiniować operatory w klasie `Symbol`:

```
# Dodanie operatorów [] i []= do klasy Symbol, które dają dostęp i pozwalają ustawiać
# metody singletonowe obiektów. Znak : należy czytać jako „metoda”, a [] odpowiada na pytanie „czego”.
# Zatem :m[o] należy czytać "metoda m obiektu o".
#
class Symbol
  # Zwraca obiekt klasy Method obiektu obj wyznaczonego przez ten symbol. Może to być metoda singletonowa
  # obiektu obj (jak metoda klasowa) lub metoda obiekta zdefiniowana
  # przez obj.class lub odziedziczona po nadklassie.
  # Przykłady:
  #   creator = :new[Object] # Metoda klasowa Object.new.
  #   doubler = :*[2]       # Metoda * obiektu klasy Fixnum 2.
  #
  def [](obj)
    obj.method(self)
  end

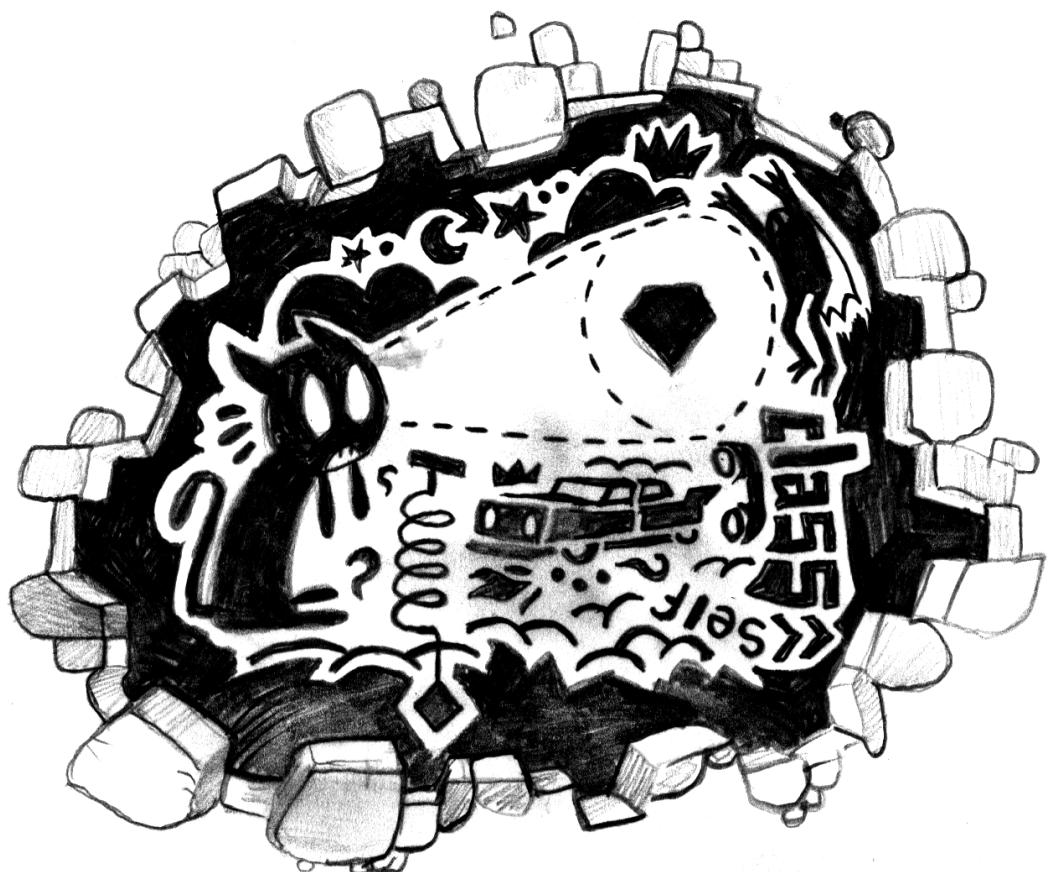
  # Definicja metody singletonowej dla obiektu o przy użyciu obiektu klasy Proc lub Method f jako jej ciała.
  # Ten symbol służy jako nazwa metody.
  # Przykłady:
  #
  # :singleton[o] = lambda { puts "to jest metoda singletonowa obiektu o" }
  # :class_method[String] = lambda { puts "to jest metoda klasowa" }
  #
  # Zauważ, że nie można utworzyć w ten sposób metody obiektowej. Zobacz Module.[]=
  #
  def []=(o, f)
    # W poniższym bloku nie można użyć obiektu self, ponieważ jest wyznaczany w
    # kontekście innego obiektu. W związku z tym self musi zostać przypisany do zmiennej.
    sym = self
    # To jest obiekt, dla którego definiujesz metody singletonowe.
    eigenclass = (class << o; self end)
    # Metoda define_method jest prywatna. Aby ją uruchomić, konieczne jest użycie metody instance_eval.
    eigenclass.instance_eval { define_method(sym, f) }
  end
end
```

Mając zdefiniowaną metodę `Symbol[]` i opisany wcześniej moduł `Functional`, można napisać sprytny (i nieczytelny) kod jak ten poniżej:

```
dashes = :*[ '-' ]          # Metoda * obiektu '>'.
puts dashes[10]              # Drukuje "-----".
y = (:+[1]*:[2])[x]          # Inny sposób zapisu  $y = 2*x + 1$ .
```

Definicja operatora `[]`= dla klasy `Symbol` przypomina definicję operatora `[]=` dla klasy `Module` w tym, że w obu przypadkach metoda `define_method` jest wywoływana za pomocą metody `instance_eval`. Różnica polega na tym, że metody singletonowe w przeciwieństwie do metod obiektowych nie są definiowane w klasach, ale w klasie *eigenklass* obiektów. Okreściecie *eigenclass* pojawi się jeszcze w rozdziale 7.

Klasy i moduły



Ruby jest językiem czysto obiektowym — każda wartość w tym języku jest obiektem (lub przy najmniej tak się zachowuje). Każdy obiekt jest egzemplarzem jakiejś klasy. Klasa definiuje zestaw metod, na które odpowiada obiekt. Klasy mogą rozszerzać inne klasy, czyli być podklasami innych klas, oraz dziedziczyć i przesłaniać metody swoich nadkлас. Klasy mogą również zawierać, aścielj mówiąc, dziedziczyć metody modułów.

Obiekty w języku Ruby sąścielj hermetyczne — dostęp do ich stanu można uzyskać wyłącznie za pomocą definiowanych przez nie metod. Do zmiennych obiektowych modyfikowanych przez te metody nie ma bezpośredniego dostępu z zewnątrz obiektu. Możliwe jest zdefiniowanie metod pobierających i ustawiających wartości, które wydają się uzyskiwać bezpośredni dostęp do stanu obiektu. Te pary metod dostępowych nazywane są **atrybutami** i są one oddzielone od zmiennych obiektowych. Metody zdefiniowane w klasie mogą być publiczne, chronione lub prywatne, co ma wpływ na to, jak i gdzie można je wywoływać.

W przeciwnieństwie do hermetycznego stanu obiektów klasy w języku Ruby są otwarte. Każdy program może dodawać metody do istniejących klas. Możliwe jest nawet dodawanie metod singletonowych do konkretnych obiektów.

Znacząca część architektury obiektowej języka Ruby wchodzi w skład jego rdzenia. Inne czynności, jak tworzenie atrybutów i deklarowanie widoczności metod, są wykonywane za pomocą metod, a nie prawdziwych słów kluczowych języka. Niniejszy rozdział zaczyna się od rozszerzonego kursu definiowania klas i dodawania do nich metod. Dalej znajdują się podrоздziały poświęcone bardziej zaawansowanym zagadnieniom:

- widoczności metod,
- tworzeniu podklas i dziedziczeniu,
- tworzeniu obiektów i ich inicjalizacji,
- modułom jako przestrzeniom nazw i dającym się dołączać do klas domieszkom,
- metodom singletonowym i klasie eigenclass,
- algorytmowi rozstrzygania nazw metod,
- algorytmowi rozstrzygania nazw stałych.

7.1. Definiowanie prostej klasy

Omawianie klas rozpocznie się od rozszerzonego kursu, w którym zbudowana zostanie klasa o nazwie `Point` reprezentująca punkt geometryczny o współrzędnych X i Y. Kolejne podrоздziały opisują:

- definiowanie nowej klasy;
- tworzenie egzemplarzy nowej klasy;
- pisanie metody inicjującej dla tej klasy;
- dodawanie atrybutowych metod dostępowych do klasy;
- definiowanie operatorów dla klasy;
- definiowanie metod iteracyjnych i sprawianie, aby klasa umożliwiała iterację;
- przesłanianie ważnych metod klasy `Object` — na przykład `to_s`, `==`, `hash` i `<=>`;
- definiowanie metod klasowych, zmiennych klasowych, zmiennych egzemplarzy klasy oraz stałych.

7.1.1. Tworzenie klasy

Do tworzenia klas służy słowo kluczowe `class`:

```
class Point  
end
```

Podobnie jak większość konstrukcji w języku Ruby, definicja klasy kończy się słowem kluczowym `end`. Poza nowymi klasami słowo kluczowe `class` tworzy nowe stałe odwołujące się do tych klas. Nazwa klasy i nazwa odpowiadającej jej stałej są takie same, dlatego wszystkie nazwy klas muszą zaczynać się od wielkiej litery.

W ciele klasy, ale poza wszelkimi metodami egzemplarzowymi zdefiniowanymi w tej klasie, słowo kluczowe `self` odnosi się do definiowanej klasy.

Jak większość instrukcji w języku Ruby `class` jest wyrażeniem. Wartością wyrażenia `class` jest wartość ostatniego wyrażenia znajdującego się w ciele klasy. Zazwyczaj ostatnim wyrażeniem w klasie jest instrukcja `def` definiująca jakąś metodę. Wartością instrukcji `def` jest zawsze `nil`.

7.1.2. Tworzenie egzemplarza klasy

Mimo iż w klasie `Point` nic jeszcze nie ma, można już utworzyć jej egzemplarz:

```
p = Point.new
```

Stała `Point` przechowuje obiekt klasy reprezentujący nową klasę. Wszystkie obiekty klas udostępniają metodę o nazwie `new` tworzącą nowe egzemplarze.

Nie można jeszcze zrobić nic ciekawego z nowo utworzonym obiektem klasy `Point`, który został zapisany w zmiennej lokalnej `p`, ponieważ w klasie tej nie zostały jeszcze zdefiniowane żadne metody. Można natomiast sprawdzić, jakiego rodzaju obiektem jest obiekt `p`:

```
p.class      #=> Point.  
p.is_a? Point #=> true.
```

7.1.3. Inicjalizacja obiektu

Tworząc nowy obiekt klasy `Point`, należy go zainicjować za pomocą dwóch liczb odpowiadających współrzędnym X i Y. W wielu obiektowych językach programowania do tego celu służy specjalny „konstruktor”, w Ruby natomiast — metoda `initialize`:

```
class Point  
  def initialize(x, y)  
    @x, @y = x, y  
  end  
end
```

W tych zaledwie trzech nowych wierszach kodu jest kilka ważnych rzeczy do omówienia. Słowo kluczowe `def` zostało szczegółowo opisane w rozdziale 6. Rozdział ten jednak był skoncentrowany na definiowaniu funkcji globalnych, których można używać w dowolnym miejscu w programie. Kiedy słowo kluczowe `def` zostaje użyte w taki sposób z niekwalifikowaną nazwą metody wewnętrz definicji klasy, definiuje **metodę obiektową** (egzemplarzy) tej klasy. Metoda obiektowa może być wywoływana na rzecz obiektów swojej klasy. Kiedy wywoływana jest metoda obiektowa, wartość `self` jest egzemplarzem klasy, w której ta metoda jest zdefiniowana.

Kolejna ważna rzecz do zapamiętania to fakt, że metoda `initialize` ma w języku Ruby specjalne przeznaczenie. Metoda `new` tworzy nowy egzemplarz klasy, a następnie automatycznie wywołuje na jego rzecz metodę `initialize`. Wszystkie argumenty przekazane do metody `new` są przesyłane do metody `initialize`. Ponieważ `initialize` wymaga dwóch argumentów, w wywołaniu metody `Point.new` należy podać dwie wartości:

```
p = Point.new(0,0)
```

Poza tym że jest automatycznie wywoływana przez metodę `Point.new`, metoda `initialize` jest automatycznie kwalifikowana jako prywatna. Obiekt może ją wywołać na samym sobie, ale nie można wywołać tej metody jawnie na rzecz jakiegoś obiektu w celu ponownego ustawienia jego stanu.

Przyjrzyj się teraz ciału metody `initialize`. Pobiera ona dwie przekazane do niej wartości zapisane w zmiennych lokalnych `x` i `y` i przypisuje je do zmiennych egzemplarza `@x` i `@y`. Zmienne egzemplarza (obiektowe) zawsze zaczynają się od znaku `@` i zawsze „należą” do tego, do czego odnosi się `self`. Każdy egzemplarz klasy `Point` posiada własną kopię tych dwóch zmiennych, które wyznaczają wartości współrzędnych `X` i `Y`.

Hermetyzacja zmiennych egzemplarza

Do zmiennych egzemplarza można uzyskać dostęp wyłącznie poprzez metody tego obiektu. Kod nieznajdujący się w metodzie obiektowej nie może odczytywać ani ustawiać wartości zmiennych tego obiektu (chyba że użyto w nim technik refleksji opisanych w rozdziale 8.).

Na koniec jeszcze jedno ostrzeżenie dla programistów przyzwyczajonych do Javy i spokrewnionych z nią języków. W językach typowanych statycznie zmienne, w tym zmienne obiektowe, muszą być zadeklarowane. Wiadomo już, że w języku Ruby zmienne nie muszą być zadeklarowane, ale niektórym może się wydawać, że konieczne jest pisanie kodu podobnego do poniższego:

```
# Niepoprawny kod!
class Point
  @x = 0    # Utworzenie zmiennej egzemplarza @x i przypisanie jej domyślnej wartości. Źle!
  @y = 0    # Utworzenie zmiennej egzemplarza @y i przypisanie jej domyślnej wartości. Źle!
  def initialize(x,y)
    @x, @y = x, y    # Inicjalizacja utworzonych wcześniej zmiennych @x i @y.
  end
end
```

Niniejszy kod nie działa dokładnie tak, jak spodziewałby się programista Javy. Zmienne egzemplarza są zawsze rozstrzygane w kontekście obiektu `self`. Kiedy wywoływana jest metoda `initialize`, `self` zawiera egzemplarz klasy `Point`. Jednak kod znajdujący się poza tą metodą jest wykonywany jako część definicji klasy `Point`. Kiedy zostaną wykonane dwa pierwsze przypisania, `self` odnosi się do samej klasy `Point`, nie do jej egzemplarza. Zmienne `@x` i `@y` znajdujące się w metodzie `initialize` nie mają nic wspólnego ze zmiennymi o tych samych nazwach poza tą metodą.

7.1.4. Definiowanie metody `to_s`

Praktycznie każda definiowana klasa powinna zawierać metodę egzemplarza `to_s` zwracającą łańcuchową reprezentację jej obiektów. Ta funkcja jest nie do przecenienia podczas usuwania błędów. Poniżej znajduje się przykładowa definicja takiej metody w klasie `Point`:

```
class Point
  def initialize(x,y)
    @x, @y = x, y
  end
  def to_s          # Zwraca łańcuch reprezentujący ten punkt.
    "(#{@x},#{@y})"  # Interpolacja zmiennych egzemplarza do łańcucha.
  end
end
```

Dzięki zdefiniowaniu tej metody można tworzyć i drukować punkty:

```
p = new Point(1,2)  # Utworzenie nowego obiektu klasy Point.
puts p              # Drukuje (1,2).
```

7.1.5. Akcesory i atrybuty

W klasie `Point` znajdują się dwie zmienne egzemplarza. Jak wiadomo, dostęp do ich wartości można uzyskać wyłącznie poprzez inne metody tego egzemplarza. Aby użytkownicy klasy `Point` mogli używać współrzędnych X i Y punktu, konieczne jest dostarczenie im metod dostępowych (akcesorów) zwracających wartości tych zmiennych:

```
class Point
  def initialize(x,y)
    @x, @y = x, y
  end
  def x           # Metoda dostępu do (czyli getter) zmiennej @x.
    @x
  end
  def y           # Metoda dostępu do zmiennej @y.
    @y
  end
end
```

Mając zdefiniowane te metody, można napisać poniższy kod:

```
p = Point.new(1,2)
q = Point.new(p.x*2, p.y*3)
```

Wyrażenia `p.x` i `p.y` mogą wyglądać jak odwołania do zmiennych, ale w rzeczywistości są to wywołania metod z opuszczonymi nawiasami.

Jeśli klasa `Point` miałaby umożliwiać modyfikowanie jej obiektów (co zazwyczaj nie jest dobrym pomysłem), należałoby również utworzyć metody ustawiające (settery) wartości zmiennych egzemplarza:

```
class MutablePoint
  def initialize(x,y); @x, @y = x, y; end
  def x; @x; end           # Metoda dostępu do zmiennej @x.
  def y; @y; end           # Metoda dostępu do zmiennej @y.
  def x=(value)            # Metoda ustawiająca wartość zmiennej @x.
    @x = value
  end
  def y=(value)            # Metoda ustawiająca wartość zmiennej @y.
    @y = value
  end
end
```

Przypomnijmy, że do wywoływania takich metod ustawiających można używać wyrażeń przypisania. W związku z tym, mając zdefiniowane powyższe metody, można napisać:

```
p = Point.new(1,1)
p.x = 0
p.y = 0
```

Używanie metod ustawiających w klasach

Po zdefiniowaniu w klasie metody ustawiającej typu `x=` kuszące może być używanie jej w innych metodach obiektowych tej samej klasy. To znaczy zamiast `@x=2` można napisać `x=2`, chcąc jawnie wywołać `x=(2)` na rzecz `self`. To oczywiście nie zadziała, ponieważ `x=2` jest zwykłym utworzeniem nowej zmiennej lokalnej.

Błąd ten jest bardzo powszechny wśród początkujących programistów języka Ruby uczących się posługiwania metodami ustawiającymi i przypisaniami. Zasada jest taka, że wyrażenia przypisania wywołują metody ustawiające tylko wówczas, gdy są wywoływane przez obiekty. Aby użyć metody ustawiającej wewnętrz klasy, w której została ona zdefiniowana, należy ją wywołać bezpośrednio poprzez `self.x=2`. Na przykład `self.x=2`.

Kombinacje zmiennych egzemplarza z prostymi metodami dostępowymi i ustawiającymi są tak powszechnne, że wymyślono sposób na automatyzację ich tworzenia. Metody `attr_reader` i `attr_accessor` są zdefiniowane w klasie `Module`. Wszystkie klasy są modułami (klasa `Class` jest podklassą klasy `Module`), dzięki czemu metody te można wywoływać w definicji każdej klasy. Obie wymienione metody przyjmują dowolną liczbę symboli reprezentujących nazwy atrybutów. Metoda `attr_accessor` tworzy gettery i settery. Rzadko używana metoda `attr_writer` tworzy tylko metody ustawiające. Aby zatem zdefiniować dającą się modyfikować klasę `Point`, można napisać:

```
class Point
  attr_accessor :x, :y # Definicja metod dostępowych dla zmiennych egzemplarza.
end
```

Natomiast niedająca się modyfikować wersja tej samej klasy wyglądałaby następująco:

```
class Point
  attr_reader :x, :y   # Definicja metod sprawdzających dla zmiennych egzemplarza.
end
```

Każda z tych metod przyjmuje również nazwy atrybutów w postaci łańcuchów zamiast symboli. Przyjęło się stosowanie symboli, ale poniższy kod również jest poprawny:

```
attr_reader "x", "y"
```

Metoda `attr` jest podobna do poprzednich, ale działa nieco inaczej w Ruby 1.8 i Ruby 1.9. W Ruby 1.8 metoda `attr` może zdefiniować tylko jeden atrybut za jednym razem. Kiedy zostanie przekazany do niej tylko symbol, definiuje ona metodę pobierającą. Jeśli po symbolu zostanie podana wartość `true`, definiuje także metodę ustawiającą:

```
attr :x           # Definicja prostej metody pobierającej x dla @x.
attr :y, true    # Definicja metody sprawdzającej i ustawiającej dla @y.
```

W Ruby 1.9 metody `attr` można używać w taki sam sposób jak w Ruby 1.8 lub jako synonim metody `attr_reader`.

Metody `attr`, `attr_reader` i `attr_accessor` tworzą metody egzemplarza. Wykonywane przez nie działania nazywa się **metaprogramowaniem**, które jest bardzo potężną właściwością języka Ruby. Więcej przykładów metaprogramowania przedstawionych jest w rozdziale 8.

Należy zauważać, że metoda attr i związane z nią pozostałe metody są wywoływanie wewnątrz definicji klasy, ale poza wszelkimi definicjami metod. Są one wykonywane tylko jeden raz, podczas definiowania klasy. Nie są związane z nimi żadne obciążenia wydajnościowe. Metody, które tworzą, są tak samo szybkie jak metody pisane ręcznie. Należy pamiętać, że mogą one tworzyć tylko proste gettery i settery, które bezpośrednio odwzorowują się na wartości zmiennych egzemplarza o takich samych nazwach. Aby utworzyć bardziej skomplikowane metody dostępowe, jak settery ustawiające zmienne o innych nazwach lub gettery zwracające wartości obliczone z dwóch różnych zmiennych, należy napisać ich kod własnoręcznie.

7.1.6. Definiowanie operatorów

Chcesz, aby operator + wykonywał działanie wektorowego dodawania dwóch obiektów klasy Point, operator * mnożył punkt przez liczbę skalarną, a jednoargumentowy operator - wykonywał działanie odpowiadające mnożeniu przez -1? Operatory metodowe takie jak + są zwykłymi metodami, których nazwy składają się ze znaków interpunkcyjnych. Ponieważ operator - występuje zarówno w wersji jedno-, jak i dwuargumentowej, nazwa metody jednoargumentowego operatora - to -@. Poniżej znajduje się klasa Point ze zdefiniowanymi operatorami matematycznymi:

```
class Point
  attr_reader :x, :y    # Definicja metod dostępowych.
  def initialize(x,y)
    @x,@y = x, y
  end
  def +(other)          # Definicja operatora + wykonującego dodawanie wektorowe.
    Point.new(@x + other.x, @y + other.y)
  end
  def -@                # Definicja jednoargumentowego operatora - negującego obie współrzędne.
    Point.new(-@x, -@y)
  end
  def *(scalar)         # Definicja operatora * wykonującego mnożenie skalarne.
    Point.new(@x*scalar, @y*scalar)
  end
end
```

Przyjrzyj się ciału metody +. Można w nim używać zmiennej obiektowej @x obiektu self, na rzecz którego metoda jest wywoływana. Nie może natomiast używać zmiennej @x innych obiektów klasy Point. W Ruby nie istnieje składnia pozwalająca na takie działanie. Wszystkie odwołania do zmiennych egzemplarza niejawnie wykorzystują obiekt self. Dlatego metoda + jest uzależniona od metod pobierających x i y (później dowiesz się, że można ograniczyć widoczność metod, dzięki czemu obiekty jednej klasy mogą używać swoich metod nawzajem, ale kod znajdujący się poza klasą nie).

Należy zauważać, że metoda * wymaga operandu liczbowego, nie obiektu klasy Point. Jeśli p jest punktem, można napisać p*2. Nie można natomiast w tej chwili napisać 2*p. To drugie wyrażenie wywołuje metodę * z klasy Integer, która nie potrafi obsłużyć obiektów klasy Point. Ponieważ klasa Integer nie dysponuje informacjami na temat mnożenia punktów, prosi punkt o pomoc, wywołując jego metodę coerce (więcej szczegółów na ten temat można znaleźć w podrozdziale 3.8.7.4). Aby wyrażenie 2*p zwracało ten sam wynik co wyrażenie p*2, można zdefiniować metodę coerce:

```
# Jeżeli obiekt klasy Point zostanie przekazany do metody * klasy Integer, metoda ta
# zostanie wywołana na rzecz tego obiektu Point, a następnie nastąpi próba pomnożenia elementów tablicy.
# Zamiast konwertować typy, zmieniasz kolejność operandów, dzięki czemu
```

```
# wywoływana jest metoda *zdefiniowana powyżej.
def coerce(other)
  [self, other]
end
```

Sprawdzanie typów i kacze typowanie

Metoda `+` w żaden sposób nie sprawdza typów, zakładając, że przekazywane są do niej zawsze właściwe obiekty. W języku Ruby często bardzo luźno podchodzi się do znaczenia słowa „właściwy”. W przypadku metody `+` właściwe są wszystkie obiekty udostępniające metody o nazwach `x` i `y`, jeśli metody te nie wymagają żadnych argumentów i zwracają jakiś rodzaj liczb. Nieważne, czy argument rzeczywiście **jest** punktem, dopóki wygląda i zachowuje się jak punkt. To podejście jest czasami nazywane kaczym typowaniem zgodnie z powiedzeniem: „jeśli chodzi jak kaczka i kwacze jak kaczka, to musi to być kaczka”.

Jeśli do metody `+` zostanie przekazany niewłaściwy obiekt, Ruby zgłosi wyjątek. Stanie się to na przykład wtedy, gdy do punktu spróbujesz dodać 3:

```
NoMethodError: undefined method `x' for 3:Fixnum
from ./point.rb:37:in `+'
```

Powyższy komunikat informuje, że obiekt `Fixnum` 3 nie udostępnia metody o nazwie `x` oraz że błąd ten powstał w metodzie `+` klasy `Point`. Informacje te w zupełności wystarczają do znalezienia źródła problemu, aczkolwiek są nieco niejasne. Sprawdzenie klasy argumentów metody może ułatwić zlokalizowanie błędu w kodzie wykorzystującym tę metodę. Poniżej znajduje się nowa wersja tej metody z weryfikacją klasy:

```
def +(other)
  raise TypeError, "Wymagany argument klasy Point" unless other.is_a? Point
  Point.new(@x + other.x, @y + other.y)
end
```

Poniżej znajduje się luźniejsza wersja sprawdzania typów z ulepszonym komunikatem, ale nadal pozwalająca na kacze typowanie:

```
def +(other)
  raise TypeError, "Wymagany argument typu Point" unless
    other.respond_to? :x and other.respond_to? :y
  Point.new(@x + other.x, @y + other.y)
end
```

Należy zauważyć, że ta wersja metody nadal zakłada, iż metody `x` i `y` zwracają liczby. Gdyby któraś z nich zwróciła na przykład łańcuch, znowu pojawiłby się niejasny komunikat o błędzie.

Inny sposób sprawdzania typów jest stosowany po fakcie. Można po prostu obsłużyć wszystkie wyjątki, które mają miejsce podczas wykonywania metody, i zgłosić bardziej odpowiedni własny wyjątek:

```
def +(other)          # Zakładasz, że other wygląda jak Point.
  Point.new(@x + other.x, @y + other.y)
rescue               # Jeśli coś powyżej pójdzie źle,
  raise TypeError, # zgłaszasz własny wyjątek.
    "Dodawanie punktów, z których jeden nie kwacze jak punkt!"
end
```

7.1.7. Dostęp do tablic za pomocą metody []

W języku Ruby dostęp do tablic jednowymiarowych i asocjacyjnych uzyskuje się za pomocą nawiasów kwadratowych. Każda klasa może mieć zdefiniowaną metodę [] i używać tych nawiasów samodzielnie. Oto definicja metody [] dla klasy Point, która będzie pozwalała na traktowanie obiektów tej klasy jako tablic tylko do odczytu o długości 2 lub jako tablic asocjacyjnych tylko do odczytu z kluczami :x i :y:

```
# Definicja metody [] pozwalającej na traktowanie obiektów klasy Point jako tablic jednowymiarowych
# lub tablic asocjacyjnych z kluczami :x i :y.
def [](index)
  case index
  when 0, -2: @x          # Indeks 0 (lub -2) odpowiada współrzędnej X.
  when 1, -1: @y          # Indeks 1 (lub -1) odpowiada współrzędnej Y.
  when :x, "x": @x        # Klucze tablicy asocjacyjnej jako symbol lub łańcuch dla współrzędnej X.
  when :y, "y": @y        # Klucze tablicy asocjacyjnej jako symbol lub łańcuch dla współrzędnej Y.
  else nil                # Tablice jednowymiarowe i asocjacyjne zwracają wartość nil dla nieprawidłowych indeksów.
  end
end
```

7.1.8. Enumeracja współrzędnych

Jeśli obiekt klasy Point może zachowywać się jak tablica zawierająca dwa elementy, powinno dać się iterować po tych elementach tak jak w prawdziwych tablicach. Poniżej znajduje się definicja iteratora each klasy Point. Dzięki temu że obiekty klasy Point zawsze zawierają dokładnie dwa elementy, iterator nie musi działać na zasadzie pętli — wystarczy dwa razy wywołać instrukcję yield:

```
# Niniejszy iterator przekazuje współrzędną X do odpowiedniego bloku, następnie
# przekazuje współrzędną Y i zwraca wartość. Pozwala na przeliczenie punktu,
# jakby był tablicą zawierającą dwa elementy. Ta metoda each jest wymagana
# przez moduł Enumerable.
def each
  yield @x
  yield @y
end
```

Po zdefiniowaniu niniejszego iteratora można pisać następujące procedury:

```
p = Point.new(1,2)
p.each { |x| print x }    # Drukuje 12.
```

Co ważniejsze, dzięki zdefiniowaniu iteratora each możliwe stało się domieszanie metod modułu Enumerable, które są zdefiniowane na podstawie iteratora each. Klasa zyskuje ponad 20 iteratorów po dodaniu tylko jednego wiersza kodu:

```
include Enumerable
```

Dzięki temu można pisać taki ciekawy kod jak poniższy:

```
# Czy punkt P oznacza początek układu współrzędnych?
p.all? { |x| x == 0 } # Prawda, jeśli zwraca true dla wszystkich elementów.
```

7.1.9. Porównywanie punktów

Przy obecnej definicji klasy dwa osobne punkty nigdy nie są równe, nawet jeśli ich współrzędne X i Y są takie same. Aby naprawić tę sytuację, należy zaimplementować operator ==(aby odświeżyć sobie wiadomości na temat różnych pojęć równości w języku Ruby, można przeczytać podrozdział 3.8.5).

```

def ==(o)
  if o.is_a? Point      # Czy self == o?
    @x==o.x && @y==o.y # Jeżeli o jest obiektem klasy Point,
  else                   # porównywane są pola.
    false                # Jeżeli o nie jest obiektem klasy Point,
  end                   # to zgodnie z definicją self != o.
end

```

Kacze typowanie a równość

Zdefiniowany wcześniej operator + w żaden sposób nie sprawdza typów — działa z wszystkimi obiektami, które udostępniają metody x i y zwracające liczby. Metoda == została zaimplementowana inaczej. Zamiast kaczego typowania wymaga, aby argumenty należały do klasy Point. Jest to wybór implementacyjny. W powyższej implementacji metody == obiekt nie może być równy punktowi, jeśli sam nie należy do klasy Point.

Inne implementacje mogą być bardziej lub mniej restrykcyjne. W powyższej implementacji klasa argumentu jest sprawdzana za pomocą predykatu `is_a?`. To pozwala, aby egzemplarz podklasy klasy Point był równy obiektyowi klasy Point. W bardziej restrykcyjnej implementacji użyto by metody `instance_of?`, aby odrzucić egzemplarze podklas. Podobnie w powyższej implementacji używa się operatora == do porównywania współrzędnych X i Y. W przypadku liczb pozwala on na konwersję typów, co oznacza, że punkt (1,1) jest równy punktowi (1.0, 1.0). Jest to najpewniej dobry sposób porównywania, ale w bardziej restrykcyjnej wersji można by było do porównywania współrzędnych użyć metody `eql?`.

Bardziej liberalna definicja równości pozwalałaby na kacze typowanie. Wymagana jest tu jednak pewna doza ostrożności. Metoda == nie powinna zgłaszać wyjątku `NoMethodError`, jeśli obiekt podany jako argument nie udostępnia metod x i y. W zamian powinien zwracać wartość `false`:

```

def ==(o)
  @x == o.x && @y == o.y      # Czy self == o?
rescue                         # Założenie, że o udostępnia odpowiednio metody x i y.
  false                          # Jeżeli założenie okaże się nieprawdziwe,
end                             # self != o.

```

Przypomnij sobie z podrozdziału 3.8.5, że obiekty Ruby udostępniają również metodę `eql?` do porównywania. Domyślnie metoda ta, podobnie jak operator ==, sprawdza identyczność obiektów, a nie porównuje ich zawartości. Często potrzebne jest, aby metoda `eql?` działała tak samo jak operator ==. Można ją do tego zmusić, tworząc alias:

```

class Point
  alias eql? ==
end

```

Z drugiej strony są dwa powody, dla których można wymagać, aby metoda `eql?` działała inaczej niż operator ==. Po pierwsze, metoda `eql?` w niektórych klasach wykonuje bardziej restrykcyjne porównywanie niż operator ==. Na przykład w klasie `Numeric` i jej podklasach operator == pozwala na konwersję typów, a metoda `eql?` nie. Jeśli uznasz, że użytkownicy klasy Point mogą potrzebować porównywania jej obiektów na dwa różne sposoby, możesz pójść tym śladem. Ponieważ punkty składają się tylko z dwóch liczb, dobrym rozwiązaniem byłoby skorzystanie z przykładu klasy `Numeric`. Metoda `eql?` wyglądałaby bardzo podobnie do metody ==, ale porównywałaby współrzędne za pomocą metody `eql?` zamiast operatora ==:

```

def eql?(o)
  if o.instance_of? Point
    @x.eql?(o.x) && @y.eql?(o.y)
  else
    false
  end
end

```

Na marginesie warto zauważyc, że takie podejście jest właściwe dla wszystkich klas implementujących kolekcje (zbiory, listy, drzewa) dowolnych obiektów. Operator == powinien porównywać elementy kolekcji za pomocą ich operatorów ==, a metoda eql? powinna do porównywania używać metody eql? tych obiektów.

Druga sytuacja, w której można potrzebować różnych implementacji metody eql? i operatora ==, to przypadek, gdy chcesz, aby egzemplarze klasy zachowywały się w specjalny sposób, kiedy są używane jako klucze w tablicach asocjacyjnych. Metoda eql? w klasie Hash porównuje klucze (nie wartości). Jeśli pozostanie ona niezdefiniowana, w tablicach asocjacyjnych egzemplarze klasy będą porównywane pod względem identyczności. Oznacza to, że jeśli z kluczem p zostanie związana jakaś wartość, to będzie ją można pobrać tylko za pomocą dokładnie tego samego obiektu p. Obiekt q nic nie zdała, nawet jeśli p == q. Obiekty, które można modyfikować, nie sprawdzają się dobrze jako klucze w tablicach asocjacyjnych, ale pozostawienie metody eql? bez definicji pozwala elegancko ominąć ten problem (więcej informacji na temat tablic asocjacyjnych i modyfikowalnych kluczy znajduje się w podrozdziale 3.4.2).

Ze względu na to, iż metoda eql? jest używana razem z tablicami asocjacyjnymi, nigdy nie należy implementować jej samej. Jeśli zdefiniuje się metodę eql?, należy również zdefiniować metodę hash obliczającą kod mieszający (ang. *hashcode*) dla obiektów. Jeśli dwa obiekty według metody eql? są równe, ich metody hash **muszą** zwracać te same wartości (dwa obiekty, które nie są sobie równe, także mogą zwrócić takie same same kody mieszające, ale należy tego unikać, jeśli to tylko możliwe).

Implementacja optymalnych metod hash może być trudna. Na szczęście istnieje prosty sposób na obliczenie idealnie odpowiednich kodów mieszających dla każdej klasy — wystarczy połączyć kody mieszające wszystkich obiektów, do których odwołuje się dana klasa (mówiąc ściślej, należy połączyć kody mieszające wszystkich obiektów porównywanych przez własną metodę eql?). Sztuka polega na połączeniu tych kodów mieszających w odpowiedni sposób. Poniższa metoda hash **nie** jest dobra:

```

def hash
  @x.hash + @y.hash
end

```

Wada tej metody jest taka, że zwraca ona ten sam kod mieszający zarówno dla punktu (1,0), jak i (0,1). Jest to dozwolone, ale niekorzystnie odbija się na szybkości, kiedy punkty są używane jako klucze tablic asocjacyjnych. Zamiast tego powinieneś nieco zamieszać:

```

def hash
  code = 17
  code = 37*code + @x.hash
  code = 37*code + @y.hash
  # Dodaj wiersz tego typu dla każdej znaczącej zmiennej egzemplarza.
  code # Zwrot powstalego kodu.
end

```

Ten ogólny przepis na kod mieszający powinien wystarczyć w większości klas Ruby. Przepis ten i stałe 17 i 37 zostały zaadaptowane z książki *Effective Java* autorstwa Joshua Blocha wydanej przez wydawnictwo Prentice Hall.

7.1.10. Porządkowanie punktów

Załóżmy, że chcesz zdefiniować jakiś porządek dla obiektów klasy `Point`, aby móc je porównywać i sortować. Punkty w układzie współrzędnych można porównywać na wiele różnych sposobów. Tutaj będą one ustawiane według odległości od początku układu. Odległość (lub wartość bezwzględna) oblicza się za pomocą twierdzenia Pitagorasa: pierwiastek kwadratowy z sumy kwadratów współrzędnych X i Y.

Aby zdefiniować taki porządek dla obiektów klasy `Point`, należy tylko zdefiniować operator `<=>` (zobacz podrozdział 4.6.6) i dodać moduł `Comparable`. Dzięki temu zostaną wzmieszone implementacje operatorów porównywania i relacyjnych opartych na operatorze `<=>`, który został zdefiniowany. Operator `<=>` powinien porównywać obiekt `self` z obiektem do niego przekazywanym. Jeśli `self` jest mniejszy niż ten przekazany obiekt (w tym przypadku znajduje się bliżej początku układu), operator powinien zwrócić wartość `-1`. Jeżeli obiekty są równe, powinna zostać zwrócona wartość `0`. Natomiast w sytuacji gdy `self` jest większy od argumentu, powinna zostać zwrócona wartość `1` (jeśli obiekt przekazany jako argument i `self` nie mogą być porównywane, powinna zostać zwrócona wartość `nil`). Poniżej znajduje się implementacja operatora `<=>`. Należy w niej zwrócić uwagę na dwie rzeczy. Po pierwsze, nie użyto metody `Math.sqrt`, a w zamian porównywane są sumy kwadratów współrzędnych. Po drugie, po obliczeniu sum kwadratów dalsze działania zostają oddelegowane do operatora `<=>` klasy `Float`:

```
include Comparable    # Domieszanie metod z modułu Comparable.  
# Definicja porządku punktów na podstawie ich odległości od początku układu współrzędnych.  
# Ta metoda jest wymagana przez moduł Comparable.  
def <=>(other)  
  return nil unless other.instance_of? Point  
  @x**2 + @y**2 <=> other.x**2 + other.y**2  
end
```

Należy zauważyć, że moduł `Comparable` definiuje metodę `==`, która wykorzystuje definicję operatora `<=>`. Operator porównywania punktów pod względem odległości od środka układu współrzędnych spowodował powstanie metody `==`, która uznaje za równe punkty `(0,1)` i `(1,0)`. Ponieważ jednak klasa `Point` zawiera własną definicję metody `==`, metoda `==` modułu `Comparable` nie jest nigdy wywoływana. Najlepiej by było, gdyby definicje równości operatorów `==` i `<=>` były spójne. Nie było to możliwe w klasie `Point`, przez co operatory pozwalają na poniższe rzeczy:

```
p,q = Point.new(1,0), Point.new(0,1)  
p == q      #=> false: p nie jest równy q.  
p < q      #=> false: p nie jest mniejszy od q.  
p > q      #=> false: p nie jest większy od q.
```

W końcu warto w tym miejscu wspomnieć, że moduł `Enumerable` definiuje kilka metod, na przykład `sort`, `min` i `max`, które działają tylko z obiektami udostępniającymi operator `<=>`.

7.1.11. Modyfikowanie punktów

Opracowywana do tej pory klasa `Point` jest niemodyfikowalna. To znaczy że po utworzeniu jej obiektu nie ma żadnego publicznego API pozwalającego na modyfikację współrzędnych X i Y tego punktu. Tak też powinno być. Spróbuj jednak zejść z wyznaczonej ścieżki i zbadać kilka metod, które dodałbyś, gdybyś chciał, aby punkty były modyfikowalne.

Przede wszystkim potrzebowałbyś metod `x=` i `y=` ustawiających bezpośrednio współrzędne X i Y. Można zdefiniować je jawnie lub zamienić `attr_reader` na `attr_accessor`:

```
attr_accessor :x, :y
```

Następnie potrzebna będzie alternatywa dla operatora `+`. Jeśli będziesz dodawać współrzędne punktu q do współrzędnych punktu p, alternatywa ta, zamiast tworzyć nowy obiekt klasy `Point`, będzie odpowiednio modyfikować punkt p. Metodę tę nazwiemy `add!`. Wykrzyknik oznacza, że metoda ta zmienia wewnętrzny stan obiektu, na rzecz którego jest wywoływana:

```
def add!(p)          # Dodaje p do self i zwraca zmodyfikowany obiekt self.  
  @x += p.x  
  @y += p.y  
  self  
end
```

Do nazwy metody zmieniającej stan obiektu dodajesz wykrzyknik tylko wtedy, gdy istnieje jej wersja niezmieniająca stanu obiektu. W tym przypadku nazwa `add!` ma sens tylko wtedy, jeśli zdefiniujesz także metodę `add` zwracającą nowy obiekt, a niezmieniającą swojego odbiorcy. Niemodyfikująca wersja metody modyfikującej często tworzy kopię obiektu `self` i wywołuje na jej rzecz swój modyfikujący odpowiednik:

```
def add(p)          # Niemodyfikująca wersja metody add!  
  q = self.dup      # Utworzenie kopii obiektu self.  
  q.add!(p)         # Wywołanie metody modyfikującej na rzecz powstalej kopii.  
end
```

W tym prostym przykładzie metoda `add` działa dokładnie tak samo jak operator `+`, który został już zdefiniowany wcześniej. Dlatego definicja tej metody jest w zasadzie niepotrzebna. W związku z tym, jeśli nie zdefiniujesz niemodyfikującej metody `add`, powinieneś rozważyć możliwość opuszczenia znaku wykrzyknika w metodzie modyfikującej. Można sprawić, aby sama nazwa (`add` zamiast `plus`) wskazywała, że metoda ta modyfikuje stan obiektów.

7.1.12. Szybkie i łatwe modyfikowalne klasy

Jednym ze sposobów na utworzenie modyfikowalnej klasy `Point` jest użycie do tego celu klasy `Struct`. Jest to rdzenna klasa języka Ruby, która służy do generowania innych klas; udostępniają one metody dostępowe do wyznaczonych przez programistę pól. Istnieją dwa sposoby na utworzenie nowej klasy za pomocą wywołania `Struct.new`:

```
Struct.new("Point", :x, :y)  # Tworzy nową klasę Struct::Point.  
Point = Struct.new(:x, :y)    # Tworzy nową klasę, przypisuje do Point.
```

Nazywanie klas anonimowych

Drugi wiersz zaprezentowanego fragmentu kodu odkrywa bardzo ciekawą własność klas w języku Ruby. Jeśli obiekt nienazwanej klasy zostanie przypisany do stałej, jej nazwa staje się nazwą tej klasy. To samo zachowanie można zaobserwować przy użyciu konstruktora `Class.new`:

```
C = Class.new    # Nowa klasa bez ciała przypisana do stałej.  
c = C.new        # Utworzenie egzemplarza tej klasy.  
c.class.to_s     # => "C": nazwa stałej staje się nazwą klasy.
```

Klas utworzonych za pomocą konstruktora Struct.new można używać tak samo jak wszystkich innych klas. Metoda new takiej klasy wymaga wartości dla każdego z nazwanych pól, które zostaną wyznaczone, a metody egzemplarza będą umożliwiać odczyt i zapis tych pól:

```
p = Point.new(1, 2)      # => #<struct Point x=1, y=2>.  
p.x                      # => 1.  
p.y                      # => 2.  
p.x = 3                  # => 3.  
p.x                      # => 3.
```

Struktury definiują również operatory [] i []= służące do indeksowania tablic zwykłych i asocjacyjnych, a także udostępniają iteratory each i each_pair służące do iteracji przez wartości zapisane w egzemplarzu struktury:

```
p[:x] = 4                # => 4: to samo co p.x =.  
p[:x]                    # => 4: to samo co p.x.  
p[1]                     # => 2: to samo co p.y.  
p.each {|c| print c}    # Drukuje "42".  
p.each_pair { |n,c| print n,c } # Drukuje "x4y2".
```

Klasy oparte na strukturach dysponujące działającym operatorem == mogą być używane jako klucze w tablicach asocjacyjnych (należy jednak zachować ostrożność, ponieważ są modyfikowalne), a nawet udostępniają przydatną metodę to_s:

```
q = Point.new(4, 2)  
q == p                # => true.  
h = {q => 1}           # Utworzenie tablicy asocjacyjnej przy użyciu q jako klucza.  
h[p]                  # => 1: pobranie wartości przy użyciu klucza q.  
q.to_s                # => "#<struct Point x=4, y=2>".
```

Klasa Point zdefiniowana jako struktura nie udostępnia metod specyficznych dla punktów, jak zdefiniowana wcześniej metoda add! czy zdefiniowany operator <=>. Natomiast nic nie stoi na przeszkodzie, aby je dodać. Definicje klas w języku Ruby nie są statyczne. Każdą klasę (w tym klasy zdefiniowane za pomocą wywołania Struct.new) można otworzyć i dodać do niej metody. Poniżej zaprezentowana została klasa Point zdefiniowana jako struktura, do której dodano metody specyficzne dla punktów:

```
Point = Struct.new(:x, :y)      # Utworzenie nowej klasy i przypisanie jej do nazwy Point.  
class Point  
  def add!(other)            # Otwarcie klasy Point w celu dodania nowych metod.  
    self.x += other.x  
    self.y += other.y  
    self  
  end  
  include Comparable          # Dodanie modułu do klasy.  
  def <=>(other)             # Definicja operatora <=>.  
    return nil unless other.instance_of? Point  
    self.x**2 + self.y**2 <=> other.x**2 + other.y**2  
  end  
end
```

Jak napisaliśmy na początku niniejszego podrozdziału, klasa Struct została zaprojektowana, aby tworzyć modyfikowalne klasy. Jednak przy odrobinie wysiłku można sprawić, że klasa oparta na Struct będzie niemodyfikowalna:

```
Point = Struct.new(:x, :y)      # Definicja modyfikowalnej klasy.  
class Point  
  undef x=, y=, []=            # Otwarcie klasy.  
  # Usunięcie definicji metod modyfikujących.  
end
```

7.1.13. Metoda klasowa

Spróbuj innego sposobu dodawania do siebie obiektów klasy `Point`. Zamiast wywoływać metodę egzemplarza na rzecz jednego punktu i przekazywać drugi punkt do tej metody, napisz metodę `sum` przyjmującą dowolną liczbę obiektów klasy `Point`, dodającą je i zwracającą nowy obiekt klasy `Point`. Nie będzie to metoda egzemplarza wywoływana na rzecz obiektu klasy `Point`, a **metoda klasowa** wywoywana przez samą klasę `Point`. Wywołanie metody `sum` może wyglądać następująco:

```
total = Point.sum(p1, p2, p3) # p1, p2 i p3 to obiekty klasy Point.
```

Nie zapomnij, że wyrażenie `Point` odnosi się do obiektu klasy `Class` reprezentującego klasę `Point`. Aby zdefiniować metodę klasową dla klasy `Point`, w rzeczywistości definiujesz singletonową metodę obiektu klasy `Point` (metody singletonowe opisaliśmy w podrozdziale 6.1.4). Do zdefiniowania metody singletonowej używa się jak zawsze słowa kluczowego `def`, ale określa się także obiekt, na którym metoda ta ma być zdefiniowana, oraz jej nazwę. Definicja metody klasowej `sum` wygląda następująco:

```
class Point
  attr_reader :x, :y      # Definicja metod dostępowych do zmiennych egzemplarza.
  def Point.sum(*points) # Zwrot sumy dowolnej liczby punktów.
    x = y = 0
    points.each { |p| x += p.x; y += p.y }
    Point.new(x,y)
  end
  # ... reszta klasy pominięta...
end
```

W tej definicji metody klasowej nazwa klasy jest wymieniona jawnie, a składnia przypomina tę używaną do wywoływania metod. Metody klasowe można także definiować za pomocą `self` zamiast nazwy klasy. W związku z tym metodę tę można również zapisać następująco:

```
def self.sum(*points) # Zwraca sumę dowolnej liczby punktów.
  x = y = 0
  points.each { |p| x += p.x; y += p.y }
  Point.new(x,y)
end
```

Użycie `self` zamiast nazwy `Point` sprawia, że kod jest nieco mniej przejrzysty, ale jest to za-stosowanie zasady DRY (ang. *Don't Repeat Yourself* — nie powtarzaj się). Jeśli zostanie użyte słowo `self` zamiast nazwy klasy, będzie ją można zmienić bez konieczności edytowania definicji jej metod klasowych.

Istnieje jeszcze jedna technika pozwalająca definiować metody klasowe. Mimo iż jest nieco mniej przejrzysta niż ta zaprezentowana wcześniej, może okazać się przydatna przy definio-waniu wielu metod klasowych oraz istnieje duże prawdopodobieństwo spotkania jej w ist-niejącym już kodzie:

```
# Otwarcie obiektu klasy Point, aby dodać do niego metody.
class << Point      # Składnia pozwalająca dodawać metody do pojedynczych obiektów.
  def sum(*points) # Metoda klasowa Point.sum.
    x = y = 0
    points.each { |p| x += p.x; y += p.y }
    Point.new(x,y)
  end
  # Tutaj mogą znajdować się definicje innych metod klasowych.
end
```

Techniki tej można także używać wewnątrz definicji klasy, gdzie, zamiast powtarzać nazwę klasy, można użyć `self`:

```
class Point
  # Metody egzemplarza.
  class << self
    # Metody klasowe.
  end
end
```

Więcej informacji na temat tej składni znajduje się w podrozdziale 7.7.

7.1.14. Stałe

Niektóre klasy mogą zyskać na wartości dzięki definicji w nich różnych stałych. Oto kilka stałych, które mogą być przydatne w klasie `Point`:

```
class Point
  def initialize(x,y)  # Inicjalizacja metody.
    @x,@y = x, y
  end
  ORIGIN = Point.new(0,0)
  UNIT_X = Point.new(1,0)
  UNIT_Y = Point.new(0,1)
  # Pozostała część definicji klasy.
end
```

Wewnątrz definicji klasy do stałych tych można odwoływać się za pomocą samych ich nazw. Poza klasą natomiast nazwy stałych muszą być poprzedzane przedrostkiem w postaci nazwy klasy, do której należą:

```
Point::UNIT_X + Point::UNIT_Y  # => (1,1)
```

Należy zauważyć, że ponieważ stałe w tym przykładzie odnoszą się do egzemplarzy klasy, nie można zdefiniować ich wcześniej niż po zdefiniowaniu metody `initialize` tej klasy. Ponadto pamiętaj, iż stałe klasy `Point` można bez żadnego problemu definiować poza definicją tej klasy:

```
Point::NEGATIVE_UNIT_X = Point.new(-1,0)
```

7.1.15. Zmienne klasowe

Zmienne klasowe są widoczne i współdzielone przez metody klasowe i metody egzemplarza klasy oraz samą definicję klasy. Podobnie do zmiennych egzemplarza zmienne klasowe są hermetyzowane — można ich używać w implementacji klasy, ale są one niewidoczne dla jej użytkowników. Nazwy zmiennych klasowych zaczynają się od znaków `@@`.

W klasie `Point` nie ma potrzeby używania zmiennych klasowych. Jednak na potrzeby nauki założymy, że chcesz zbierać informacje na temat liczby utworzonych obiektów klasy `Point` i ich średnich współrzędnych. Poniżej przedstawiony został odpowiedni przykładowy kod:

```
class Point
  # Inicjacja zmiennych klasowych w definicji klasy.
  @@n = 0          # Ile zostało utworzonych punktów.
  @@totalX = 0    # Suma wszystkich współrzędnych X.
  @@totalY = 0    # Suma wszystkich współrzędnych Y.
  def initialize(x,y) # Metoda initialize.
    @x,@y = x, y   # Ustawienie wartości początkowych dla zmiennych egzemplarza.
    # Użycie zmiennych klasowych w tej metodzie egzemplarza do zbierania danych.
  end
end
```

```

@@n += 1           # Słedzenie liczby utworzonych punktów.
@@totalX += x     # Dodanie współrzędnych do sum całkowitych.
@@totalY += y
end
# Metoda klasowa raportująca o zebranych danych.
def self.report
    # Użycie zmiennych klasowych w metodzie klasowej.
    puts "Liczba utworzonych punktów: #{@@n}"
    puts "Średnia współrzędna X: #{@@totalX.to_f/@@n}"
    puts "Średnia współrzędna Y: #{@@totalY.to_f/@@n}"
end

```

W powyższym kodzie warto zauważać, że zmienne klasowe są wykorzystywane w metodach egzemplarza, metodach klasowych i w samej definicji klasy, poza wszystkimi metodami. Zmienne klasowe różnią się od zmiennych egzemplarza w podstawowych kwestiach. Wiadomo, że zmienne egzemplarza są zawsze wyznaczane w odniesieniu do obiektu `self`. Dlatego właśnie odwołanie do zmiennej egzemplarza w definicji klasy lub metodzie klasowej jest całkiem inne niż odwołanie do zmiennej egzemplarza w metodzie egzemplarza. Zmienne klasowe są natomiast zawsze wyznaczane w odniesieniu do obiektu klasy utworzonego przez otaczającą instrukcję `class`.

7.1.16. Klasowe zmienne egzemplarza

Klasy są obiektami, a więc mogą jak wszystkie obiekty posiadać zmienne egzemplarza. Klasowe zmienne egzemplarza nie są tym samym co zmienne klasowe. Są do nich jednak na tyle podobne, że można ich używać zamiennie.

Zmienna egzemplarza użyta wewnętrz definiacji `class`, ale poza metodami egzemplarza, jest klasową zmienną egzemplarza. Podobnie jak zmienne klasowe, klasowe zmienne egzemplarza są związane z klasą, a nie z jakimś konkretnym jej obiektem. Wadą klasowych zmiennych egzemplarza jest to, że nie można ich używać wewnętrz metod egzemplarza tak jak zmiennych klasowych. Bez prefiksów w postaci znaków interpunkcyjnych może być trudno zapamiętać, czy dana zmienna jest związana z egzemplarzami klasy, czy z obiektem klasowym.

Jedna z największych zalet klasowych zmiennych egzemplarza w stosunku do zmiennych klasowych ma związek z niejasnym zachowaniem tych drugich podczas tworzenia podklasy istniejącej klasy. Wróćmy do tego później.

Przekonwertuj zbierającą statystyki klasę `Point`, aby zamiast zmiennych klasowych używała klasowych zmiennych egzemplarza. Jedyna trudność polega na tym, że ponieważ klasowe zmienne egzemplarza nie mogą być używane w metodach egzemplarza, kod odpowiedzialny za statystyki należy przenieść poza metodę `initialize` (która jest metodą egzemplarza) do metody klasowej `new` służącej do tworzenia punktów:

```

class Point
    # Inicjacja klasowych zmiennych egzemplarza w samej definicji klasy.
    @@n = 0           # Ile zostało utworzonych punktów.
    @@totalX = 0     # Suma współrzędnych X.
    @@totalY = 0     # Suma współrzędnych Y.
    def initialize(x,y) # Metoda initialize.
        @x,@y = x, y # Ustawienie początkowych wartości zmiennych egzemplarza.
    end
    def self.new(x,y) # Metoda klasowa tworząca obiekty klasy Point.
        # Klasowe zmienne egzemplarza do przechowywania danych.
        @@n += 1         # Słedzenie liczby utworzonych obiektów klasy Point.
    end

```

```

@totalX += x      # Dodanie współrzędnych do sum całkowitych.
@totalY += y
super             # Wywołanie prawdziwej definicji new w celu utworzenia punktu.
                  # Więcej na temat słowa kluczowego super znajduje się dalej w tym rozdziale.
end
# Metoda klasowa raportująca zebrane dane.
def self.report
  # Użycie klasowych metod egzemplarza w metodzie klasowej.
  puts "Liczba utworzonych punktów: #{@n}"
  puts "Średnia współrzędna X: #{@totalX.to_f#{@n}}"
  puts "Średnia współrzędna Y: #{@totalY.to_f#{@n}}"
end

```

Ponieważ klasowe zmienne egzemplarza są tylko zmiennymi egzemplarza obiektu klasowego, do utworzenia ich metod dostępowych można używać metod attr, attr_reader i attr_accessor. Sztuka polega na wywołaniu tych metod metaprogramowania w odpowiednim miejscu. Przypomnijmy, że jednym ze sposobów na zdefiniowanie metody klasowej jest użycie składni class << self. Ta sama składnia pozwala na definiowanie metod dostępu do atrybutów dla klasowych zmiennych egzemplarza:

```

class << self
  attr_accessor :n, :totalX, :totalY
end

```

Po zdefiniowaniu tych akcesorów można odwoływać się do surowych danych jako Point.n, Point.totalX i Point.totalY.

7.2. Widoczność metod — publiczne, chronione i prywatne

Metody egzemplarza mogą być **publiczne** (`public`), **prywatne** (`private`) lub **chronione** (`protected`). Programiści znający jakiś inny obiektowy język programowania najprawdopodobniej znają te pojęcia. Jednak należy uważnie przeczytać niniejszy podrozdział, ponieważ słowa te w języku Ruby mają nieco inne znaczenie niż w innych językach.

Metody są domyślnie publiczne, chyba że zostaną jawnie zadeklarowane jako prywatne lub chronione. Jednym z wyjątków jest metoda `initialize`, która jest zawsze prywatna. Innym wyjątkiem są metody globalne zadeklarowane poza wszystkimi klasami — takie metody są prywatnymi metodami egzemplarza klasy `Object`. Metodę publiczną można wywołać w dowolnym miejscu — nie ma żadnych ograniczeń dotyczących jej stosowania.

Metoda prywatna jest przeznaczona do użytku wewnętrz klasy i może być wywoływana tylko przez inne metody egzemplarza tej samej klasy (lub, jak przekonasz się później, jej podklas). Metody prywatne są niejawnie wywoływanie na rzecz `self` i nie można ich jawnie wywoływać na rzecz obiektów. Jeśli `m` jest metodą prywatną, musi być wywoływana w **stylu funkcyjnym** jako `m`. Nie można napisać `o.m` ani nawet `self.m`.

Metoda chroniona przypomina metodę prywatną pod tym względem, że może być wywoływana wyłącznie w klasie lub jej podklasach. Różnica pomiędzy tymi dwiema metodami polega na tym, że metodę chronioną można wywołać jawnie na rzecz dowolnego egzemplarza klasy i nie jest ona ograniczona do niejawnych wywołań na rzecz `self`. Metody chronionej

można na przykład użyć do zdefiniowania metody dostępowej pozwalającej egzemplarzom klasy na współdzielenie stanu wewnętrznego, ale niepozwalającej użytkownikom tej klasy na dostęp do tego stanu.

Metody chronione są najrzadziej używane z wszystkich dostępnych rodzajów, a dodatkowo najtrudniej je zrozumieć. Zasadę, kiedy można wywołać metodę chronioną, można formalnie ująć w następujący sposób: metoda chroniona zdefiniowana w klasie C może zostać wywołana na rzecz obiektu o przez metodę w obiekcie p, jeśli klasy obiektów o i p są podklasami klasy C lub są jej równe.

Do określania widoczności metod służą trzy metody o nazwach `public`, `private` i `protected`. Są to metody egzemplarza klasy `Module`. Wszystkie klasy są modułami i wewnątrz definicji klasy (ale na zewnątrz definicji metod) słowo `self` odnosi się do definiowanej klasy. W związku z tym `public`, `private` i `protected` można używać samodzielnie jako słów kluczowych. W rzeczywistości są to jednak wywołania metod na rzecz `self`. Istnieją dwa sposoby na wywołanie tych metod. Jeśli w wywoaniu nie zostaną podane żadne argumenty, wszystkie znajdujące się poniżej definicje metod będą miały określoną widoczność. W klasie można używać ich następująco:

```
class Point
  # Metody publiczne.
  # Poniżej znajdują się metody chronione.
  protected
    # Metody chronione.
    # Poniższe metody są prywatne.
  private
    # Metody prywatne.
end
```

Metody te można także wywoływać przy użyciu nazwy jednej lub więcej metod (jako symboli lub łańcuchów) jako argumentów. Kiedy są wywoywane w taki sposób, zmieniają widoczność metod o podanych nazwach. W takim przypadku deklaracja widoczności musi znajdować się za definicją metody. Jednym ze sposobów jest deklarowanie wszystkich prywatnych i chronionych metod w jednym miejscu na końcu definicji klasy. Inne podejście polega na deklarowaniu widoczności każdej chronionej lub prywatnej metody bezpośrednio po jej zdefiniowaniu. Poniżej na przykład znajduje się klasa zawierająca prywatną metodę użytkową i chronioną metodę dostępową:

```
class Widget
  def x                      # Metoda dostępu do @x.
    @x
  end
  protected :x                # Deklaracja chronienia.
  def utility_method          # Definicja metody.
    nil
  end
  private :utility_method     # Deklaracja prywatności.
end
```

Należy pamiętać, że metody `public`, `private` i `protected` mają zastosowanie tylko do metod. Zmienne egzemplarza i klasowe są hermetyzowane, a więc w efekcie prywatne. Stałe są natomiast publiczne. Nie da się sprawić, aby zmienna egzemplarza była dostępna na zewnątrz klasy (oczywiście poza zdefiniowaniem metody dostępowej). Nie da się również zdefiniować stałej, która byłaby niedostępna na zewnątrz klasy.

Czasami dobrze jest zadeklarować metodę klasową jako prywatną. Jeśli w klasie znajdują się na przykład metody fabryki, może okazać się potrzebne zadeklarowanie metody `new` jako prywatnej. Do tego celu należy użyć metody `private_class_method`, podając przynajmniej jedną nazwę metody w postaci symbolu:

```
private_class_method :new
```

Aby prywatną metodę klasową uczynić z powrotem publiczną, można użyć metody `public_class_method`. Żadna z powyższych metod nie może zostać wywołana bez argumentów w taki sposób jak metody `public`, `protected` i `private`.

Ruby jest w założeniu projektowym bardzo otwartym językiem programowania. Możliwość oznaczania wybranych metod jako prywatnych i chronionych pozwala na stosowanie dobrego stylu programowania i zapobiega niezamierzonemu użyciu metod, które nie wchodzą w skład publicznego API klasy. Ważne jest jednak, aby pamiętać, że dzięki metaprogramowaniu w języku Ruby można z łatwością wywołać prywatną lub chronioną metodę, a nawet uzyskać dostęp do hermetycznych zmiennych egzemplarza. Do wywołania prywatnej metody użytkowej zdefiniowanej wcześniej można użyć metody `send` lub ewaluować blok w kontekście obiektu za pomocą metody `instance_eval`:

```
w = Widget.new                      # Utworzenie widgetu.  
w.send :utility_method                # Wywołanie prywatnej metody!  
w.instance_eval { utility_method }    # Inny sposób wywołania tej metody.  
w.instance_eval { |x| }                # Odczyt zmiennej egzemplarza w.
```

Aby wywołać metodę po jej nazwie, ale nie chcąc przypadkowo wywołać jakieś prywatnej metody, o której nie wiesz, można (w Ruby 1.9) zamiast metody `send` użyć metody `public_send`. Działa ona podobnie jak `send`, ale nie wywołuje metod prywatnych. Metody `public_send`, `send` i `instance_eval` zostały opisane w rozdziale 8.

7.3. Tworzenie podklas i dziedziczenie

W większości obiektowych języków programowania, także w Ruby, dostępny jest mechanizm **tworzenia podklas**. Pozwala on na tworzenie nowych klas, które bazują na istniejących już klasach, ale są ich zmodyfikowanymi wersjami. Na początek wyjaśnienie podstawowej terminologii związanej z tym zagadniением. Programiści Javy, C++ lub innego podobnego języka powinni już te określenia znać.

Definiując klasę, można zaznaczyć, że **rozszerza** ona inną klasę zwaną **nadklassą** — lub **dziedziczy** po niej. Jeśli klasa Ruby rozszerza klasę `Gem`, mówi się, że Ruby jest **podklassą** klasy `Gem`, a klasa `Gem` jest **nadklassą** klasy Ruby. Jeśli podczas definicji klasy nie zostanie podana nazwa nadklasy, nowa klasa rozszerza klasę `Object`. Każda klasa może mieć dowolną liczbę podklas i jedną nadkласę, z wyjątkiem klasy `Object`, która nie ma nadklastry.

Dzięki temu że klasy mogą mieć wiele podklas i tylko jedną nadklastry, można je przedstawić w postaci drzewa zwanego **hierarchią klas** Ruby. Korzeniem tego drzewa jest klasa `Object` i wszystkie pozostałe klasy dziedziczą bezpośrednio lub pośrednio po niej. **Potomkowie** klasy to podklasy tej klasy, podklasy tych podklas itd. **Przodkowie** klasy to jej nadklasa, nadklasa jej nadklastry itd. aż do klasy `Object`. Rysunek 5.5 w rozdziale 5. przedstawia część hierarchii klas Ruby obejmującą klasę `Exception` i wszystkich jej potomków. Na rysunku tym widać, że przodkami klasą `EOFError` są klasy `IOError`, `StandardError`, `Exception` i `Object`.

Klasa BasicObject w Ruby 1.9

W języku Ruby 1.9 klasa `Object` nie jest już korzeniem hierarchii klas. Jest nim nowa klasa o nazwie `BasicObject`, a klasa `Object` jest jej podklasą. Klasa `BasicObject` jest bardzo prosta, ma bardzo mało własnych metod i jest przydatna jako nadklasa delegacyjnych klas osłonowych (jak zaprezentowana na listingu 8.5 w rozdziale 8.).

Nowo tworzone klasy w języku Ruby 1.9 nadal rozszerzają klasę `Object`, chyba że wyraźnie zostanie to zaznaczone inaczej. Większość programistów nigdy nie będzie potrzebować ani rozszerzać klasę `BasicObject`. Metody takie jak `==`, `equal?`, `instance_eval` oraz `_send_` zwykle uważa się za metody klasy `Object`, choć w rzeczywistości zdefiniowane są w klasie `BasicObject`.

Składnia służąca do rozszerzania klas jest prosta. Wystarczy do instrukcji `class` dodać znak `<` i nazwę rozszerzanej klasy. Na przykład:

```
class Point3D < Point      # Definicja klasy Point3D jako podklasy klasy Point.  
end
```

W kolejnych podrozdziałach zostanie rozwinięta ta trójwymiarowa klasa `Point`. Na tym przykładzie zostanie zademonstrowane dziedziczenie metod z nadklas oraz przesłanianie metod i ich wzbogacanie w podklasach w celu uzyskania w nich nowych funkcji.

Tworzenie podklasy klasy Struct

We wcześniejszej części niniejszego rozdziału zobaczyłeś, jak za pomocą metody `Struct.new` automatycznie wygenerować prostą klasę. Możliwe jest utworzenie podklasy takiej klasy, dzięki czemu możliwe jest dodanie metod innych niż te wygenerowane automatycznie:

```
class Point3D < Struct.new("Point3D", :x, :y, :z)  
  # Nadklasa przekazała metody ==, to_s itd.  
  # Tutaj można dodać kolejne metody specyficzne dla punktów 3D.  
end
```

7.3.1. Dziedziczenie metod

Klasa `Point3D`, która została zdefiniowana wcześniej, jest prostą podklasą klasy `Point`. Deklaruje, że jest rozszerzeniem klasy `Point`, ale ponieważ nie posiada żadnego ciała, nic się do tej klasy nie dodaje. W efekcie obiekt klasy `Point3D` jest tym samym co obiekt klasy `Point`. Jedyna różnica, którą można zauważyci pomiędzy tymi klasami, to wartość zwracana przez metodę `class`:

```
p2 = Point.new(1,2)  
p3 = Point3D.new(1,2)  
print p2.to_s, p2.class    # Drukuję "(1,2)Point".  
print p3.to_s, p3.class    # Drukuję "(1,2)Point3D".
```

Wartość zwracana przez metodę `class` jest oczywiście inna, ale uderzające jest to, co nie ulega zmianie. Obiekty klasy `Point3D` dziedziczą metodę `to_s` zdefiniowaną w klasie `Point`. Ponadto odziedziczona została także metoda `initialize` — umożliwia ona tworzenie obiektów `Point3D` za pomocą takiego samego wywołania metody `new` jak przy tworzeniu obiektów klasy

Point¹. W kodzie tym znajduje się jeszcze jeden przykład dziedziczenia — zarówno klasa Point, jak i Point3D dziedziczą metodę class po klasie Object.

7.3.2. Przesłanianie metod

Definiując nową klasę, dodaje się do niej nowe rodzaje zachowań w postaci definicji metod. Równie ważne jest dostosowywanie odziedziczonych rodzajów zachowań w klasie poprzez ponowne zdefiniowanie odziedziczonych metod.

Na przykład klasa Object definiuje bardzo ogólną metodę to_s konwertującą obiekty nałańcuchy:

```
o = Object.new
puts o.to_s      # Drukuję coś w rodzaju "#<Object:0xb7f7fce4>".
```

Definiując metodę to_s w klasie Point, w rzeczywistości **przesłoniłeś** metodę to_s odziedziczoną po klasie Object.

Jedną z najważniejszych rzeczy, które należy zrozumieć w programowaniu obiektowym i tworzeniu podklas, jest to, że wywołana metoda jest poszukiwana dynamicznie, dzięki czemu udaje się znaleźć odpowiednią definicję lub przeddefiniowaną wersję. To oznacza, że wywołania metod nie są statycznie związane w chwili ich przetwarzania, ale są dynamicznie odszukiwane, kiedy trzeba je wykonywać. Poniżej znajduje się przykład demonstrujący tę ważną kwestię:

```
# Powitanie ze światem.
class WorldGreeter
  def greet
    puts "#{greeting} #{who}"          # Wyświetlenie powitania.
  end
  def greeting                         # Wybór powitania.
    "Witaj"
  end
  def who                             # Kogo powitać.
    "Świecie"
  end
end
# Powitanie ze światem po hiszpańsku.
class SpanishWorldGreeter < WorldGreeter
  def greeting                         # Przesłonięcie przywitania.
    "Hola"
  end
end
# Wywołujemy metodę zdefiniowaną w klasie WorldGreeter, która wywołuje przesloniętą
# wersję powitania w klasie SpanishWorldGreeter oraz drukuje „Hola świecie”.
SpanishWorldGreeter.new.greet
```

Dla osób znających obiektowy styl programowania powyższy przykład jest oczywisty i dziennie prosty. Dla nowicjuszy natomiast może zawierać dużo nowych rzeczy. Wywołana została metoda greet odziedziczona po klasie WorldGreeter. Wywołała ona metodę greeting. W chwili gdy została zdefiniowana metoda greet, metoda greeting zwracałałałańcuch „Witaj”. Jednak metoda WorldGreeter została rozszerzona i obiekt, na rzecz którego wywołana została metoda greet, posiadał nową definicję metody greeting. Kiedy została wywołana

¹ Może to być zaskakujące dla programistów Javy. W klasach w tym języku definiowane są specjalne metody zwane konstruktorami, które służą do inicjacji. Metody te nie są dziedziczone. W języku Ruby metoda initialize jest zwykłą metodą dziedziczoną jak wszystkie inne.

metoda `greeting`, interpreter odnalazł odpowiednią definicję tej metody dla obiektu, na rzecz którego została wywołana. Dzięki temu na końcu uzyskane zostało hiszpańskie powitanie zamiast polskiego. Taki proces odnajdywania definicji metody w czasie działania programu nazywa się **rozstrzyganiem nazwy metody**. Został on szczegółowo opisany w podrozdziale 7.8.

Warto także zauważyc, że bardzo rozsądne jest zdefiniowanie **abstrakcyjnej** klasy wywołującej określone niezdefiniowane abstrakcyjne metody, których zdefiniowanie pozostawia się podklasom. Przeciwieństwem abstrakcji jest **konkretność**. Klasa rozszerzająca klasę abstrakcyjną jest konkretna, jeśli zawiera definicje wszystkich metod swoich przodków. Na przykład:

```
# Ta klasa jest abstrakcyjna. Nie definiuje metod greeting ani who.  
# Nie jest wymagana żadna specjalna składnia — każda klasa wywołująca metody  
# mające być zaimplementowane w podklasie jest abstrakcyjna.  
class AbstractGreeter  
  def greet  
    puts "#{greeting} #{who}"  
  end  
end  
# Konkretna podkласa.  
class WorldGreeter < AbstractGreeter  
  def greeting; "Witaj"; end  
  def who; "świecie"; end  
end  
WorldGreeter.new.greet # Wyświetla „Witaj świecie”.
```

7.3.2.1. Przesłanianie metod prywatnych

Prywatnych metod nie można wywoływać poza klasą, w której są zdefiniowane. Są one jednak dziedziczone przez podklasy. Oznacza to, że w podklasach również można je wywoływać i przesłaniać.

Zawsze należy uważać, jeśli robi się podkласę klasy, której nie napisało się własnoręcznie. Metody prywatne są często wykorzystywane w klasach jako wewnętrzne metody pomocnicze. Nie są one częścią publicznego API klasy i nie są przeznaczone do użytku na zewnątrz. Bez dostępu do kodu źródłowego klasy nie wiadomo nawet, jak nazywają się zdefiniowane w niej metody prywatne. Jeśli w podkласie zostanie zdefiniowana metoda (bez względu na jej widoczność) o takiej samej nazwie jak jedna z metod prywatnych w nadklassie, metoda w nadklassie zostanie nieumyślnie przesłonięta, a to z kolei prawie zawsze powoduje niezamierzone zachowania klasy.

Skutek jest taki, że w języku Ruby należy tworzyć podklasy tylko tych klas, których implementacja jest nam znana. Aby skorzystać tylko z publicznego API klasy, a nie jej implementacji, należy rozszerzyć funkcjonalność tej klasy poprzez hermetyzację i delegację zadań do niej, a nie poprzez dziedziczenie.

7.3.3. Wzbogacanie zachowań poprzez tworzenie łańcuchów

Czasami, przesłanając metodę, nie chcesz jej całkowicie zastąpić, a tylko wzbogacić jej działanie poprzez dodanie nowych procedur. Aby to zrobić, potrzebny jest sposób na wywołanie przesłoniętej metody z wnętrza metody przesłaniającej. Czynność ta nazywa się tworzeniem łańcucha wywołań i do jej wykonania służy słowo kluczowe `super`.

Działa ono jak specjalne wywołanie metody — wywołuje metodę o takiej samej nazwie jak bieżąca metoda znajdująca się w nadklasie bieżącej klasy (nadklaśa nie musi sama w sobie zawierać definicji tej metody — może dziedziczyć ją po jednym ze swoich przodków). Do słowa kluczowego `super` można podawać argumenty jak do zwykłego wywołania metody. Jednym z najważniejszych i najczęściej wykorzystywanych miejsc do tworzenia łańcuchów wywołań jest metoda `initialize`. Poniżej znajduje się przykładowa metoda `initialize` klasy `Point3D`:

```
class Point3D < Point
  def initialize(x,y,z)
    # Przekazanie dwóch pierwszych argumentów do metody initialize w nadklaśsie.
    super(x,y)
    # Trzeci argument pozostanie w podklaście.
    @z = z;
  end
end
```

Jeśli słowo kluczowe `super` zostanie użyte bez żadnych argumentów i nawiasów, wszystkie argumenty, które zostały przekazane do bieżącej metody, zostaną przesłane do tej metody w nadklaście. Należy jednak zauważyć, że do metody w nadklaście przekazywane są bieżące wartości parametrów metody. Jeśli metoda ta zmodyfikuje te wartości w swoich zmiennych parametrowych, do wywołania metody w nadklaście zostaną przekazane zmodyfikowane wartości.

Tak jak w zwykłych wywołaniach metod nawiasy otaczające argumenty słowa kluczowego `super` są opcjonalne. Ponieważ jednak gołe słowo `super` ma specjalne znaczenie, aby przekazać zero argumentów z metody, która ma jeden lub więcej argumentów, należy postawić po nim parę pustych nawiasów.

7.3.4. Dziedziczenie metod klasowych

Metody klasowe mogą być dziedziczone i przesłaniane tak samo jak metody egzemplarza. Jeśli w klasie `Point` znajduje się definicja metody `sum`, klasa `Point3D` odziedziczy tę metodę. To znaczy jeśli klasa `Point3D` nie będzie zawierała własnej definicji metody o nazwie `sum`, wyrażenie `Point3D.sum` będzie wywoływało tę samą metodę co wyrażenie `Point.sum`.

Ze względów stylistycznych metody klasowe lepiej wywoływać poprzez obiekty klas, w których zostały zdefiniowane. Osoba czytająca kod, widząc wyrażenie `Point3D.sum`, próbowałaby znaleźć definicję metody `sum` w klasie `Point3D` i mogłoby zajść jej dużo czasu, zanim znalazłyby tę definicję w klasie `Point`. Wywołując metodę klasową na rzecz jawnego odbiorcy, powinno się unikać polegania na dziedziczeniu — metody klasowe należy zawsze wywoływać poprzez klasy, w których zostały zdefiniowane².

W ciele metody klasowej można wywołać pozostałe metody klasowe tej klasy bez podawania jawnego odbiorcy — zostaną wtedy wywołane niejawnie na rzecz `self`, a wartością `self` w metodzie klasowej jest klasa, na rzecz której metoda ta jest wywoływana. To właśnie w ciele metody klasowej przydatne jest dziedziczenie metod klasowych — pozwala na niejawne wywoływanie metod klasowych, nawet jeśli metody te są zdefiniowane w nadklaście.

Na koniec warto zauważyć, że metody klasowe mogą wykorzystywać słowo kluczowe `super` tak samo jak metody egzemplarza mogą wywoływać metody o takiej samej nazwie w nadklaście.

² Wyjątkiem jest metoda `Class.new` — jest ona dziedziczona przez i wywoływana na rzecz każdej nowo zdefiniowanej klasy.

7.3.5. Dziedziczenie i zmienne egzemplarza

Zmienne egzemplarza często wydają się dziedziczone w języku Ruby. Spójrz na przykład na poniższy kod:

```
class Point3D < Point
  def initialize(x,y,z)
    super(x,y)
    @z = z;
  end
  def to_s
    "(#{@x}, #{@y}, #{@z})" # Czy zmienne @x i @y są dziedziczone?
  end
end
```

Metoda `to_s` w klasie `Point3D` odwołuje się do zmiennych `@x` i `@y` z nadklasy `Point`. Poniższy fragment kodu działa prawdopodobnie zgodnie z oczekiwaniami użytkownika:

```
Point3D.new(1,2,3).to_s #=> "(1, 2, 3)".
```

Ponieważ procedura ta działa tak jak należy, można skusić się na stwierdzenie, że wymienione zmienne są dziedziczone. Nie są one zdefiniowane w klasie obiektu — zostają po prostu utworzone w chwili przypisania do nich wartości. Ponieważ zmienne egzemplarza nie są definiowane przez klasy, nie podlegają dziedziczeniu.

W powyższym kodzie klasa `Point3D` została zdefiniowana metoda `initialize` łańcuchowo wywołującą metodę `initialize` swojej nadklasy. Ta druga z metod przypisuje wartości zmiennym `@x` i `@y`, dzięki czemu zostają one utworzone dla konkretnego egzemplarza klasy `Point3D`.

Programiści Javy i innych języków ze ścisłą kontrolą typów, w których klasa definiuje zestaw pól dla swoich egzemplarzy, będą musieli spędzić trochę czasu na przyzwyczajaniu się do takiego stylu programowania. Jest to jednak naprawdę proste — zmienne egzemplarza w języku Ruby nie są dziedziczone i nie mają nic wspólnego z mechanizmem dziedziczenia. Czasami może wydawać się, że są dziedziczone, ponieważ zmienne egzemplarza są tworzone przez te metody, które jako pierwsze przypiszą im wartości, a te **metody** są zazwyczaj dziedziczone lub wywoływane łańcuchowo.

Ma to jedno ważne następstwo. Ponieważ zmienne egzemplarza nie mają nic wspólnego z dziedziczeniem, zmenna egzemplarza użyta w podklasie nie może przesłonić zmiennej egzemplarza o takiej samej nazwie w nadklasie. Jeśli w podklasie zostanie użyta zmenna egzemplarza o takiej samej nazwie jak któraś ze zmiennych używanych przez jej przodków, zostanie zastosowana wartość tej zmiennej w podklasie. Można to zrobić celowo, aby zmienić zachowanie przodka, lub nieumyślnie. Ta druga sytuacja, jeśli ma miejsce, prawie zawsze powoduje błędy. Jest to jeszcze jeden dowód w dodatku do dziedziczenia prywatnych metod, które zostało opisane wcześniej, na to, że bezpiecznie jest rozszerzać tylko te klasy, których implementacja jest znana użytkownikowi i którą może on kontrolować.

7.3.6. Dziedziczenie a zmienne klasowe

Zmienne klasowe są współdzielone przez klasę i wszystkie jej podklasy. Jeśli w klasie `A` jest zdefiniowana zmenna `@@a`, to jej podklaśa o nazwie `B` może tej zmiennej używać. Mimo iż na pierwszy rzut oka może się wydawać, że jest to dziedziczenie, w rzeczywistości to coś innego.

Różnica staje się jasna, kiedy rozważysz ustawianie wartości zmiennej klasowej. Jeśli podklasa przypisuje wartość do zmiennej klasowej, która jest już używana przez nadkласę, nie tworzy własnej prywatnej kopii tej zmiennej, ale zmienia wartość dostępną w nadklasie. Dodatkowo zmienia wartość współdzieloną widoczną we wszystkich pozostałych podklasach tej nadklasty. Jeśli Ruby 1.8 zostanie uruchomiony z opcją `-w`, w takiej sytuacji zgłasza ostrzeżenie. Ruby 1.9 takiego ostrzeżenia nie zgłasza.

Jeśli jakaś klasa używa zmiennych klasowych, to każda jej podklasa może zmienić zachowanie jej i wszystkich jej potomków, zmieniając wartość tej współdzielonej zmiennej klasowej. Jest to mocny argument za używaniem zmiennych egzemplarza klasy zamiast zmiennych klasowych.

Poniższy kod demonstruje współdzielenie zmiennych klasowych. Drukuje 123:

```
class A
  @@value = 1
  def A.value; @@value; end      # Zmienna klasowa.
                                  # Metoda dostępu do tej zmiennej klasowej.
end
print A.value                  # Wyświetlenie wartości zmiennej klasy A.
class B < A; @@value = 2; end  # Podklasa zmienia wartość współdzielonej zmiennej klasowej.
print A.value                  # Nadklasa widzi zmienioną wartość.
class C < A; @@value = 3; end  # Inna klasa znowu zmienia współdzieloną zmienią.
print B.value                  # Pierwsza podklasa widzi wartość z drugiej podklasty.
```

7.3.7. Dziedziczenie stałych

Stałe są dziedziczone i można je przesłaniać tak samo jak metody egzemplarza. Jednak pomiędzy dziedziczeniem metod a dziedziczeniem stałych istnieje jedna ważna różnica.

Na przykład w klasie `Point3D` można używać stałej `ORIGIN` zdefiniowanej w klasie `Point`. Mimo iż najlepiej ze względów stylistycznych jest dodawać do stałych kwalifikator w postaci nazwy klasy, w której zostały one zdefiniowane, klasa `Point3D` może odwoływać się do tej stałej bez kwalifikatora, a nawet jako `Point3D::ORIGIN`.

W dziedziczeniu stałych zaczyna działać coś ciekawego, gdy są one przedefiniowywane w klasach typu `Point3D`. Trójwymiarowa klasa reprezentująca punkt powinna raczej posiadać stałą `ORIGIN` reprezentującą punkt trójwymiarowy. W związku z tym powinien znajdować się w niej następujący wiersz:

```
ORIGIN = Point3D.new(0,0,0)
```

Jak wiadomo, Ruby zgłasza ostrzeżenie, kiedy przedefiniowywana jest stała. W tym jednak przypadku tworzona jest nowa stała. Od tej pory istnieją dwie stałe — `Point::ORIGIN` i `Point3D::ORIGIN`.

Ważna różnica między stałymi a metodami jest taka, że stałych najpierw poszukuje się w zakresie leksykalnym miejsca, w którym zostały użyte, a dopiero potem w hierarchii dziedziczenia (szczegółowe informacje na ten temat znajdują się w podrozdziale 7.9). Oznacza to, że jeśli klasa `Point3D` odziedziczy metody używające stałej `ORIGIN`, zachowanie tych metod nie zmieni się, jeżeli w klasie `Point3D` zostanie zdefiniowana nowa wersja stałej `ORIGIN`.

7.4. Tworzenie i inicjacja obiektów

Obiekty w języku Ruby zazwyczaj tworzy się za pomocą metody `new`. Niniejszy podrozdział szczegółowo opisuje, jak to się odbywa, oraz traktuje o kilku innych mechanizmach (takich jak klonowanie i szeregowanie) pozwalających tworzyć obiekty. W każdym z podrozdziałów omówione są sposoby usprawnienia inicjalizacji nowo utworzonych obiektów.

7.4.1. Metody `new`, `allocate` i `initialize`

Każda klasa dziedziczy metodę klasową `new`. Jej zadanie jest podwójne — alokacja nowego obiektu (czyli jego utworzenie) oraz inicjacja nowo utworzonego obiektu. Zadania te są przez tę metodę delegowane odpowiednio do metod `allocate` i `initialize`. Gdyby metoda `new` została napisana rzeczywiście w języku Ruby, wyglądałaby mniej więcej tak:

```
def new(*args)
  o = self.allocate    # Utworzenie nowego obiektu tej klasy.
  o.initialize(*args) # Wywołanie metody initialize tego obiektu z argumentami.
  o                  # Zwrócenie nowego obiektu i zignorowanie wartości zwrotnej metody initialize.
end
```

Metoda `allocate` jest metodą egzemplarza klasy `Class`, którą dziedziczą wszystkie obiekty. Jej przeznaczeniem jest utworzenie nowego egzemplarza klasy. Można ją wywołać samodzielnie, aby utworzyć niezainicjowany obiekt dowolnej klasy. Nie należy tylko jej przesłaniać. Ruby zawsze wywołuje tę metodę bezpośrednio, ignorując wszelkie jej przesłaniające wersje w podklasach.

Metoda `initialize` jest metodą egzemplarza. Większość klas potrzebuje tej metody, a każda klasa poza klasą `Object` powinna wywoływać tę metodę łańcuchowo z nadklassy za pomocą słowa kluczowego `super`. Głównym zadaniem tej metody jest tworzenie zmiennych egzemplarza i ustawianie ich wartości początkowych. Zazwyczaj wartości tych zmiennych są pobierane z argumentów przekazanych przez kod kliencki do metody `new`, a przez tę metodę do metody `initialize`. Metoda `initialize` nie musi zwracać zainicjowanego obiektu. W rzeczywistości jej wartość zwrotna jest ignorowana. Ruby niejawnie czyni metodę `initialize` prywatną, co oznacza, że nie można jej jawnie wywołać na rzecz obiektu.

Class::new i Class#new

Klasa `Class` zawiera definicje dwóch metod o nazwie `new`. Jedna z nich o nazwie `Class#new` jest metodą egzemplarza, a druga o nazwie `Class::new` metodą klasową (używamy tu jednoznacznej konwencji nazewnictowej narzędzia ri). Pierwsza z wymienionych metod to opisana już metoda egzemplarza. Dziedziczą ją wszystkie obiekty — wtedy staje się metodą klasową ich klas i jest używana do tworzenia i inicjacji nowych egzemplarzy.

Metoda klasowa `Class::new` jest własną wersją tej metody klasy `Class` i można za jej pomocą tworzyć nowe klasy.

7.4.2. Metody fabryczne

Często dobrze jest pozwolić na tworzenie egzemplarzy klasy na kilka sposobów. Nierzadko można to zrobić poprzez dostarczenie domyślnych wartości parametrów w metodzie `initialize`. Jeśli definicja metody `initialize` jest taka jak poniżej, metodę `new` można wywołać z dwoma lub trzema argumentami:

```

class Point
  # Inicjuje obiekt klasy Point dwoma lub trzema współrzędnymi.
  def initialize(x, y, z=nil)
    @x,@y,@z = x, y, z
  end
end

```

Czasami jednak domyślne wartości dla parametrów nie wystarczą. Wtedy konieczne jest napisanie **metody fabryki** innej niż new do tworzenia egzemplarzy klasy. Założmy, że chcesz mieć możliwość inicjacji obiektów klasy Point za pomocą współrzędnych kartezjańskich i biegunkowych:

```

class Point
  # Definiowanie i inicjacja metod jak zwykle...
  def initialize(x,y)  # Przyjmuje współrzędne kartezjańskie.
    @x,@y = x,y
  end
  # Metoda fabryczna new jest prywatna.
  private_class_method :new
  def Point.cartesian(x,y)  # Metoda fabryczna dla współrzędnych kartezjańskich.
    new(x,y)  # Metodę new można nadal wywoływać w innych metodach klasowych.
  end
  def Point.polar(r, theta) # Metoda fabryczna dla współrzędnych biegunkowych.
    new(r*Math.cos(theta), r*Math.sin(theta))
  end
end

```

Niniejszy kod nadal w znacznym stopniu polega na metodach new i initialize, ale ponieważ metoda new jest prywatna, użytkownicy tej klasy nie mogą wywołać jej bezpośrednio. W zamian muszą użyć jednej z bardziej precyzyjnych metod fabrycznych.

7.4.3. Metody dup, clone i initialize_copy

Innym sposobem na tworzenie obiektów jest użycie metod dup i clone (zobacz podrozdział 3.8.8). Metody te alokują nowy egzemplarz klasy obiektu, na rzecz którego zostały wywołane. Następnie kopią wszystkie zmienne egzemplarza i informacje o tym, czy ich odbiorca jest pewny, czy nie (ang. *tainted*), do nowo utworzonego obiektu. Metoda clone idzie o krok dalej niż dup — również kopiuje metody singletonowe odbiorcy oraz zamraża skopiowany obiekt, jeśli oryginał jest zamrożony.

Jeśli klasa zawiera definicję metody o nazwie initialize_copy, metody clone i dup wywołują ją na rzecz skopiowanego obiektu po zakończeniu kopiowania zmiennych egzemplarza oryginału (metoda clone wywołuje metodę initialize_copy przed zamrożeniem kopii obiektu, dzięki czemu metoda ta może tę kopię zmodyfikować). Oryginalny obiekt jest przekazywany jako argument do metody initialize_copy, która może dokonać wszelkich zmian w obiekcie skopiowanym. Nie może jednak utworzyć swojej własnej kopii obiektu. Wartość zwrotna metody initialize_copy jest ignorowana. Podobnie jak metoda initialize, metoda initialize_copy jest zawsze prywatna.

Kiedy metody clone i dup kopią zmienne egzemplarza z oryginalnego obiektu do jego kopii, kopią referencje do wartości tych zmiennych, nie kopią rzeczywistych wartości. Innymi słowy, metody te wykonują kopianie płytke. Jest to jeden z powodów, dla których wiele klas może wymagać zmiany działania tych metod. Poniżej znajduje się kod definiujący metodę initialize_copy wykonującą głębsze kopianie stanu wewnętrznego:

```

class Point          # Punkt w przestrzeni n.
  def initialize(*coords) # Przyjmuje dowolną liczbę współrzędnych.
    @coords = coords      # Zapisuje współrzędne w tablicy.
  end
  def initialize_copy(orig) # Jeśli obiekt ten będzie kopowany,
    @coords = @coords.dup   # zostanie utworzona także kopia tablicy współrzędnych.
  end
end

```

Zaprezentowana tu klasa przechowuje swój wewnętrzny stan w tablicy. Bez metody `initialize_copy`, jeśli obiekt byłby kopowany za pomocą metody `dup` lub `clone`, skopiowany obiekt odwoływałby się do tej samej tablicy stanu co oryginał. Zmiany w kopii obiektu miałyby wpływ na stan oryginału. Ponieważ nie chcesz tego, musisz zdefiniować metodę `initialize_copy` tworzącą także kopię tej tablicy.

W przypadku niektórych klas, jak tych, które definiują typy wyliczeniowe, może być konieczne ograniczenie liczby egzemplarzy. W klasach takich metoda `new` musi być prywatna oraz często trzeba zabronić tworzenia kopii. Poniższy kod demonstruje jeden ze sposobów na osiągnięcie tego:

```

class Season
  NAMES = %w{ Spring Summer Autumn Winter } # Tablica pór roku.
  INSTANCES = []                            # Tablica obiektów klasy Season.
  def initialize(n) # Stan pory roku jest tylko jej
    @n = n                                # indeksem w tablicach NAMES i INSTANCES.
  end
  def to_s        # Zwrócenie nazwy pory roku.
    NAMES[@n]
  end
  # Tworzy egzemplarze tej klasy reprezentujące pory roku
  # oraz definiuje stałe odwołujące się do tych egzemplarzy.
  # Musi to zostać zrobione po zdefiniowaniu metody initialize.
  NAMES.each_with_index do |name,index|
    instance = new(index)                  # Utworzenie nowego egzemplarza.
    INSTANCES[index] = instance           # Zapisanie go w tablicy egzemplarzy.
    const_set name, instance              # Definicja stałej odwołującej się do niego.
  end
  # Kiedy zostały utworzone wszystkie egzemplarze, które będą kiedykolwiek potrzebne,
  # trzeba zablokować tworzenie kolejnych egzemplarzy.
  private_class_method :new,:allocate   # Uczynienie metod fabrycznych prywatnymi.
  private :dup, :clone                  # Uczynienie kopowania metod czynnością prywatną.
end

```

Niniejszy kod zawiera elementy metaprogramowania i będzie łatwiejszy do zrozumienia po przeczytaniu rozdziału 8. Najważniejszą częścią tego kodu jest ostatni wiersz, w którym metody `dup` i `clone` zostały zadeklarowane jako prywatne.

Innym sposobem na uniemożliwienie kopowania obiektów jest usunięcie metod `dup` i `clone` za pomocą słowa kluczowego `undef`. Można także przeddefiniować te metody w taki sposób, aby zgłaszały wyjątek informujący, że kopowanie jest niedozwolone. Taki komunikat byłby pomocny dla innych programistów korzystających z naszej metody.

7.4.4. Metody `marshal_dump` i `marshal_load`

Trzeci sposób tworzenia obiektów polega na wywołaniu metody `Marshal.load` ponownie tworzącej obiekty, które zostały wcześniej zaszeregowane (lub zserializowane) za pomocą metody `Marshal.dump`. Metoda `Marshal.dump` zapisuje klasę obiektu i rekursywnie szereguje

wartości wszystkich jej zmiennych egzemplarza. Jest to dobry sposób, ponieważ za pomocą tych dwóch metod można zapisać i przywrócić większość obiektów.

W niektórych klasach konieczne jest zmodyfikowanie sposobu szeregowania (i deszeregowania) obiektów. Jednym z powodów ku temu jest chęć dostarczenia bardziej zwięzlej reprezentacji stanu obiektu. Innym powodem jest konieczność uniknięcia zapisywania zbyt wielu danych, jak zawartość pamięci podręcznej, którą trzeba by było wyczyścić, gdyby obiekt był deszeregowany. Sposób szeregowania obiektów można zmodyfikować, definiując w klasie metodę egzemplarza o nazwie `marshal_dump`. Powinna ona zwracać inny obiekt (jak łańcuch czy tablica wybranych wartości zmiennych egzemplarza) do zaszeregowania w miejsce obiektu odbiorcy.

Jeśli zdefiniuje się metodę `marshal_dump`, konieczne jest też zdefiniowanie odpowiadającej jej metody `marshal_load`; jest ona wywoływana na rzecz nowo alokowanych (za pomocą metody `allocate`), ale niezainicjowanych egzemplarzy klasy. Należy do niej przekazywać zrekonstruowane kopie obiektów zwrócone przez metodę `marshal_dump`. Musi ona inicjować stan obiektu odbiorcy na podstawie obiektu, który został do niej przekazany.

Wróćmy na przykład do wielowymiarowej klasy `Point`, która została zdefiniowana wcześniej. Jeśli doda się ograniczenie, że wszystkie współrzędne muszą być liczbami całkowitymi, można zmniejszyć rozmiar szeregowanego obiektu o kilka bajtów, pakując tablicę współrzędnych całkowitoliczbowych do łańcucha (aby zrozumieć działanie tego kodu, dobrze jest przeczytać dokumentację ri metody `Array.pack`):

```
class Point
  # Punkt w przestrzeni n-wymiarowej.
  def initialize(*coords)
    @coords = coords
    # Przyjmuje dowolną liczbę współrzędnych.
    # Zapisuje współrzędne w tablicy.
  end
  def marshal_dump
    @coords.pack("w*")
  end
  def marshal_load(s)
    # Odpakowuje współrzędne z rozszeregowanego łańcucha
    @coords = s.unpack("w*") # i inicjuje za ich pomocą obiekt.
  end
end
```

W klasie — jak na przykład zaprezentowana wcześniej klasa `Season` — w której zostały wyłączone metody `clone` i `dup`, trzeba także zaimplementować odpowiednie metody szeregujące, ponieważ kopię obiektu można z łatwością utworzyć, szeregując go, a następnie rozszeregowując. Szeregowanie można całkowicie wyłączyć, definiując metody `marshal_dump` i `marshal_load` w taki sposób, aby zgłaszały wyjątek, ale jest to raczej ciężkie rozwiążanie. Bardziej eleganckie jest takie zmodyfikowanie deszeregowania, że metoda `Marshal.load` zwraca istniejący obiekt, zamiast tworzyć jego kopię.

W tym celu konieczne jest zdefiniowanie dodatkowej pary własnych metod szeregowania, ponieważ wartość zwrotna metody `marshal_load` jest ignorowana. Metoda o nazwie `_dump` jest metodą egzemplarza, która musi zwrócić stan obiektu w postaci łańcucha. Odpowiadająca jej metoda `_load` jest metodą klasową przyjmującą łańcuch zwrócony przez metodę `_dump` i zwracającą obiekt. Metoda `_load` może utworzyć nowy obiekt lub zwrócić referencję do istniejącego obiektu.

Aby umożliwić szeregowanie, ale zablokować kopiowanie, obiektów klasy `Season`, dodano do tej klasy poniższe metody:

```

class Season
  # Pozwalasz na szeregowanie obiektów klasy Season, ale nie
  # pozwalasz na tworzenie podczas ich rozszeregowywania nowych egzemplarzy.
  def _dump(limit)          # Metoda szeregująca.
    @n.to_s                 # Zwrot indeksu w postaciłańcucha.
  end
  def self._load(s)         # Metoda rozszeregowująca.
    INSTANCES[Integer(s)] # Zwrot istniejącego egzemplarza.
  end
end

```

7.4.5. Wzorzec Singleton

Singleton to klasa mająca tylko jeden egzemplarz. Singletony w programowaniu obiektowym mogą służyć do przechowywania globalnego stanu programu i mogą stanowić przydatną alternatywę dla metod i zmiennych klasowych.

Nomenklatura singletonowa

Niniejszy podrozdział opisuje dobrze znany w programowaniu obiektowym wzorzec Singleton. W języku Ruby trzeba uważać, kiedy używa się terminu Singleton, ponieważ został on przeciążony. Metoda dodana do pojedynczego obiektu zamiast klasy obiektów nazywa się **metodą singletonową** (zobacz podrozdział 6.1.4). Niejawny obiekt klasy, do której takie metody są dodawane, jest czasami nazywany klasą singletonową (choć w tej książce określa się ją mianem **eigenclass** — zobacz podrozdział 7.7).

Prawidłowa implementacja singletonu wymaga zastosowania kilku z opisanych wcześniej sztuczek. Metody new i allocate muszą być prywatne, metody dup i clone muszą mieć zablokowane tworzenie kopii itd. Na szczęście wszystko to robi za użytkownika dostępny w bibliotece standardowej moduł Singleton. Wystarczy dodać wiersz require 'singleton', a następnie dołączyć moduł Singleton do swojej klasy. W ten sposób zyskuje się definicję metody o nazwie instance, która nie przyjmuje żadnych argumentów i zwraca jedyny egzemplarz tej klasy. Aby zainicjować ten jedyny egzemplarz niniejszej klasy, należy zdefiniować metodę initialize. Trzeba jednak pamiętać, że do metody tej nie zostaną przekazane żadne argumenty.

W ramach przykładu wróć do klasy Point, od której zaczyna się ten rozdział, i jeszcze raz spróbuj rozwiązać problem zbierania statystyk dotyczących tworzenia punktów. Zamiast zapisywać je w zmiennych klasowych samej klasy Point, użyj singletonowego egzemplarza klasy PointStats:

```

require 'singleton'           # Moduł Singleton nie jest wbudowany.
class PointStats             # Definiowanie klasy.
  include Singleton           # Zadeklarowanie klasy jako singletonowej.
  def initialize              # Normalna metoda initialize.
    @n, @totalX, @totalY = 0, 0.0, 0.0
  end
  def record(point)           # Zapisanie nowego punktu.
    @n += 1
    @totalX += point.x
    @totalY += point.y
  end
  def report                  # Raportowanie statystyk punktów.
    puts "Liczba utworzonych punktów: #{@n}"
  end
end

```

```
    puts "Średnia współrzędna X: #{@totalX/@n}"
    puts "Średnia współrzędna Y: #{@totalY/@n}"
end
end
```

Mając taką klasę, można w klasie Point napisać następującą metodę initialize:

```
def initialize(x,y)
  @x,@y = x,y
  PointStats.instance.record(self)
end
```

Metodę klasową instance automatycznie definiuje moduł Singleton. Na rzecz utworzonego przez nią jedynego obiektu wywołuje się zwykłą metodę record. Podobnie aby sprawdzić statystyki punktów, należy wpisać:

```
PointStats.instance.report
```

7.5. Moduły

Moduł, podobnie jak klasa, jest nazwanym zestawem metod, stałych i zmiennych klasowych. Definiowanie modułów przypomina definiowanie klas z tą różnicą, że używa się słowa kluczowego module zamiast class. Moduł jednak w przeciwieństwie do klasy nie może mieć egzemplarzy ani podklas. Moduły są samodzielne — nie ma czegoś takiego jak hierarchia dziedziczenia modułów.

Modułów używa się jako przestrzeni nazw i domieszek. Oba te zastosowania są wyjaśnione w dwóch kolejnych podrozdziałach.

Podobnie jak obiekt klasy jest egzemplarzem klasy Class, obiekt modułu jest egzemplarzem klasy Module. Klasa Class jest podklassą klasy Module. Oznacza to, że wszystkie klasy są modułami, ale nie wszystkie moduły są klasami. Klas można także używać jako przestrzeni nazw, podobnie jak modułów, nie można natomiast używać ich w roli domieszek.

7.5.1. Moduły jako przestrzenie nazw

Moduły są dobrym sposobem na zgrupowanie powiązanych ze sobą metod, kiedy nie ma konieczności stosowania obiektowego stylu programowania. Założmy na przykład, że piszesz metody kodujące i dekodujące dane binarne na postać tekstową i w drugą stronę przy użyciu kodowania Base64. Nie ma potrzeby tworzenia specjalnych obiektów kodujących i dekodujących, a więc nie ma powodu do definiowania klasy. Wszystko, czego potrzebujesz, to dwie metody — jedna kodująca i druga dekodująca. Można zdefiniować tylko dwie metody globalne:

```
def base64_encode
end
def base64_decode
end
```

Aby uniknąć kolizji z innymi metodami kodującymi i dekodującymi, nazwy metod zostały poprzedzone przedrostkiem base64. Takie rozwiązanie jest możliwe, ale większość programistów woli w miarę możliwości nie dodawać metod do globalnej przestrzeni nazw. W takiej sytuacji lepiej byłoby zdefiniować te dwie metody w module o nazwie Base64:

```
module Base64
  def self.encode
end
```

```
def self.decode  
end  
end
```

Prefiks `self.` przed nazwami metod sprawia, że są one „metodami klasowymi” tego modułu. Równie dobrze można bezpośrednio wpisać nazwę modułu:

```
module Base64  
  def Base64.encode  
  end  
  def Base64.decode  
  end  
end
```

Taki sposób definiowania metod wymaga większej liczby powtórzeń, ale jest bliższy składni wywołań poniższych metod:

```
# Wywołania metod modułu Base64.  
text = Base64.encode(data)  
data = Base64.decode(text)
```

Należy pamiętać, że nazwy modułów, podobnie jak nazwy klas, muszą zaczynać się od wielkiej litery. Zdefiniowanie modułu powoduje utworzenie stałej o takiej samej nazwie jak ten moduł. Wartością tej stałej jest obiekt klasy `Module` reprezentujący ten moduł.

Moduły mogą także zawierać stałe. Implementacja modułu `Base64` mogłaby zawierać stałą przechowującą łańcuch złożony z 64 znaków wykorzystywanych jako symbole w kodowaniu `Base64`:

```
module Base64  
  DIGITS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ' \  
           'abcdefghijklmnopqrstuvwxyz' \  
           '0123456789+/'  
end
```

Poza modułem `Base64` do stałej tej można odwoływać się jako `Base64::DIGITS`. Wewnątrz modułu natomiast metody `encode` i `decode` mogą odwoływać się do niej za pomocą samej nazwy `DIGITS`. Jeśli dwie metody muszą dzielić się jakimiś niestałymi danymi, można użyć zmiennych klasowych (z prefiksem `@@`), podobnie jak w klasie.

7.5.1.1. Zagnieżdżanie przestrzeni nazw

Moduły, a także klasy, można zagnieżdżać. Skutkiem tego jest powstanie zagnieżdżonych przestrzeni nazw i nic więcej. To znaczy klasa lub moduł zagnieżdżone w innej klasie lub innym module nie mają żadnych specjalnych uprawnień dostępu do klasy lub modułu, w których zostały zagnieżdżone. Kontynuując przykład modułu `Base64`, założymy, że chcesz zdefiniować specjalne klasy kodujące i dekodujące. Ponieważ klasy `Decoder` i `Encoder` są ze sobą powiązane, zagnieźdź je w module:

```
module Base64  
  DIGITS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/'  
  class Encoder  
    def encode  
    end  
  end  
  class Decoder  
    def decode  
    end  
  end  
  # Funkcja użytkowa do dyspozycji obu klas.
```

```
def Base64.helper
end
end
```

Dzięki takiej strukturze kodu zdefiniowane zostały dwie klasy — `Base64::Encoder` i `Base64::Decoder`. Wewnątrz modułu `Base64` klasy te mogą się do siebie wzajemnie odwoływać przy użyciu swoich niekwalifikowanych nazw, czyli bez prefiksu `Base64`. W każdej z tych klas można także używać stałej `DIGITS` bez prefiksu.

Rozważmy teraz funkcję użytkową `Base64.helper`. Zagnieżdżone klasy `Encoder` i `Decoder` nie mają żadnych specjalnych uprawnień dostępu do metod zawierającego je modułu, a więc muszą odwoływać się do tej metody za pomocą jej pełnej kwalifikowanej nazwy `Base64.helper`.

Ponieważ klasy są modułami, je również można zagnieżdżać. Zagnieżdżenie jednej klasy w drugiej ma wpływ na przestrzeń nazw tylko tej wewnętrznej klasy. Nie daje to tej klasie żadnych specjalnych uprawnień dostępu do metod lub zmiennych klasy zewnętrznej. Jeśli w implementacji klasy wymagana jest klasa pomocnicza, klasa pośrednicząca lub jakaś inna klasa niewchodząca w skład publicznego API, można rozważyć zagnieżdżenie tej wewnętrznej klasy w klasie, która jej używa. Pozwala to zachować porządek w przestrzeniach nazw, ale nie powoduje, że zagnieżdżona klasa staje się w jakikolwiek sposób prywatna.

Wyjaśnienie sposobu rozstrzygania nazw stałych w zagnieżdżonych modułach znajduje się w podrozdziale 7.9.

7.5.2. Moduły jako domieszkki

Drugie zastosowanie modułów jest bardziej przydatne. Jeśli moduł zawiera definicje metod egzemplarza zamiast metod klasowych, metody te można wzmieścić w inne klasy. Powszechnie znany modułami domieszkowymi (ang. *mixins*) są `Enumerable` i `Comparable`. Moduł `Enumerable` zawiera definicje przydatnych iteratorów zaimplementowanych na podstawie iteratora `each`. Moduł `Enumerable` sam nie definiuje metody `each`, ale każda klasa, która posiada jej definicję, może domieszczać ten moduł, aby zyskać zestaw wielu przydatnych innych iteratorów. Moduł `Comparable` jest podobny — definiuje operatory porównujące na podstawie komparatora ogólnego przeznaczenia `<=>`. Jeśli klasa zawiera definicję metody `<=>`, można do niej domieszczać moduł `Comparable`, aby zyskać metody `<`, `<=`, `==>`, `>=` i `between?`.

Chcąc wzmieścić moduł w klasę, należy użyć metody `include`. Metody tej używa się zazwyczaj w taki sposób, jakby była słowem kluczowym języka:

```
class Point
  include Comparable
end
```

W rzeczywistości jest to prywatna metoda egzemplarza klasy `Module` niejawnie wywoływana na rzecz `self` — klasy, do której jest dołączany moduł. Przy użyciu formy metody powyższy kod miałby następującą postać:

```
class Point
  include(Comparable)
end
```

Ponieważ metoda `include` jest prywatna, musi być wywoływana jako funkcja, a więc nie można napisać `self.include(Comparable)`. Metoda `include` przyjmuje dowolną liczbę obiektów klasy `Module` do wzmieszania, dlatego klasy definiujące metody `each` i `<=>` mogą zawierać poniższy wiersz:

```
include Enumerable, Comparable
```

Dołączanie modułu ma wpływ na działanie metody sprawdzającej typy `is_a?` i operatora równości `==`. Na przykład klasa `String` ma domieszkę modułu `Comparable`, a w Ruby 1.8 także modułu `Enumerable`:

```
"text".is_a? Comparable          #=> true.  
Enumerable === "text"           #=> true w Ruby 1.8, false w 1.9.
```

Należy zauważać, że metoda `instanceof?` sprawdza tylko klasę swojego odbiorcy, nie nadklasy ani moduły. Dlatego poniższy kod zwraca wartość `false`:

```
"text".instance_of? Comparable #=> false.
```

Mimo że każda klasa jest modułem, metoda `include` nie pozwala na dołączanie klasy do innej klasy. Argumenty metody `include` muszą być modułami zadeklarowanymi za pomocą słowa kluczowego `module`, a nie klasami.

Można natomiast dołączyć moduł do modułu. W takim przypadku metody egzemplarza dołączanego modułu stają się metodami egzemplarza modułu biorącego. Jako przykład posłuży poniższy kod z rozdziału 5.:

```
module Iterable      #Klasy, które definiują metodę next, mogą dołączać ten moduł.  
  include Enumerable    #Iteratory są definiowane na podstawie metody each.  
  def each              #Metoda each jest zdefiniowana na podstawie metody next.  
    loop { yield self.next }  
  end  
end
```

Normalny sposób na domieszanie modułu polega na użyciu metody `Module.include`. Można to także zrobić za pomocą metody `Object.extend`. Zamienia ona metody egzemplarza wyznaczonego modułu lub wyznaczonych modułów w metody singletonowe obiektu odbiorcy (jeśli obiekt odbiorca jest egzemplarzem klasy `Class`, metody odbiorcy stają się metodami klasowymi tej klasy). Oto przykład:

```
countdown = Object.new      #Stary zwykły obiekt.  
def countdown.each           #Iterator each jako metoda singletonowa.  
  yield 3  
  yield 2  
  yield 1  
end  
countdown.extend(Enumerable) #Teraz obiekt ten posiada wszystkie metody Enumerable.  
print countdown.sort         #Drukuję "[1, 2, 3]".
```

7.5.3. Dołączalne moduły przestrzeni nazw

Możliwe jest zdefiniowanie modułu definiującego przestrzeń nazw i pozwalającego na dołączanie jego metod. Przykładem takiego modułu jest moduł `Math`:

```
Math.sin()    #=> 0.0: Math jest przestrzenią nazw.  
include Math   #Przestrzeń nazw Math można dołączyć.  
sin()         #=> 0.0: Teraz masz ułatwiony dostęp do funkcji.
```

Moduł `Kernel` również jest tego typu — można wywoływać jego metody poprzez przestrzeń nazw `Kernel` lub jako prywatne metody klasy `Object`, do której moduł `Kernel` jest dołączony.

Aby utworzyć moduł podobny do modułów `Math` i `Kernel`, należy zdefiniować swoje metody jako metody egzemplarza tego modułu. Następnie za pomocą metody `module_function` należy przekonwertować je na „funkcje modułu”. Metoda `module_function` jest prywatną

metodą egzemplarza klasy `Module`, podobną do metod `public`, `protected` i `private`. Jako argumenty przyjmuje dowolną liczbę nazw metod (w postaci symboli lub łańcuchów). Głównym efektem wywołania metody `module_function` jest to, że tworzy ona kopie wyznaczonych metod jako metody klasowe. Drugi skutek jej działania jest taki, że metody egzemplarza stają się prywatne (więcej informacji na ten temat znajduje się nieco dalej).

Metodę `module_function`, podobnie jak metody `public`, `protected` i `private`, można wywoływać bez żadnych argumentów. Przy takim wywołaniu wszystkie metody egzemplarza zdefiniowane dalej w module staną się funkcjami modułu — będą publicznymi metodami klasowymi i prywatnymi metodami egzemplarza. Raz wywołana metoda `module_function` bez argumentów działa do końca definicji modułu — w związku z tym, aby zdefiniować metody, które nie mają być funkcjami modułu, należy ich definicje umieścić wcześniej.

W pierwszej chwili może wydawać się zaskakujące to, że metoda `module_function` zamienia metody egzemplarza modułu na prywatne. Powodem tego nie jest kontrola dostępu, ponieważ metody te są oczywiście dostępne publicznie także poprzez przestrzeń nazw modułu. Metody te są zamieniane na prywatne, aby ograniczyć sposoby ich wywoływania do stylu funkcyjnego bez jawnego odbiorcy (powodem, dla którego nazywa się je **funkcjami** modułu, a nie metodami modułu, jest to, że muszą być wywoływane w stylu funkcyjnym). Dzięki wymuszeniu wywoływania funkcji dołączonego modułu bez odbiorcy zmniejsza się ryzykoomyłkowego wzięcia ich za prawdziwe metody egzemplarza. Założmy, że definiujesz klasę, której metody wykonują dużo obliczeń trygonometrycznych. Dla wygody dołączasz moduł `Math`. Dzięki temu można pisać wywołania typu `sin` zamiast `Math.sin`. Metoda `sin` jest niejawnie wywoływana na rzecz `self`, ale w rzeczywistości nie oczekujesz, że metoda ta wykonana na tym obiekcie jakieś działania.

Definiując funkcję modułu, należy unikać używania `self`, ponieważ wartość `self` zależy od sposobu wywołania tej funkcji. Z pewnością możliwe jest zdefiniowanie funkcji modułu, która działa różnie w zależności od sposobu jej wywołania. Jeśli jednak zdecydujesz się to zrobić, bardziej sensowne jest zdefiniowanie jednej metody klasowej i jednej metody egzemplarza.

7.6. Funkcje load i require

Programy Ruby można rozdzielać na wiele plików. Najbardziej naturalnym sposobem takiego podziału jest umieszczenie każdej większej klasy w osobnym pliku. Następnie wszystkie pliki można złożyć w jeden program (jeśli są dobrze zaprojektowane, mogą być wykorzystywane w wielu różnych programach) za pomocą `load` i `require` — są to globalne funkcje zdefiniowane w module `Kernel`, ale są używane jak słowa kluczowe. Ta sama metoda `require` jest także używana do ładowania plików z biblioteki standardowej.

Funkcje `load` i `require` pełnią podobne zadania, przy czym druga z nich jest znacznie częściej używana. Obie pozwalają na załadowanie i wykonanie wyznaczonego pliku z kodem źródłowym Ruby. Jeśli plik do załadowania jest wyznaczony za pomocą ścieżki bezwzględnej lub względem katalogu głównego użytkownika (`~`), wtedy ładowany jest ten określony plik. Zazwyczaj jednak plik jest wyznaczany za pomocą ścieżki względnej. Wtedy funkcje `load` i `require` szukają go względem katalogów ścieżki wczytywania Ruby (szczegółowe informacje na temat ścieżki wczytywania znajdują się dalej).

Ruby 1.9 definiuje również metodę `require_relative`. Działa tak jak `require`, z tym, że ignoruje ustawienie ścieżki wczytywania i dokonuje wyszukiwania względem ścieżki, z której został załadowany kod zawierający jej wywołanie.

Poza tymi ogólnymi podobieństwami istnieje kilka ważnych różnic między funkcjami `load` a `require`:

- Poza kodem źródłowym funkcja `require` może także wczytywać binarne rozszerzenia Ruby. Rozszerzenia binarne są oczywiście zależne od implementacji, ale w implementacjach opartych na C zazwyczaj mają postać współdzielonych plików z rozszerzeniem `.so` lub `.dll`.
- Funkcja `load` wymaga podania pełnej nazwy pliku, razem z rozszerzeniem. Funkcji `require` zazwyczaj przekazuje się nazwę biblioteki, bez rozszerzenia, zamiast nazwy pliku. W takim przypadku funkcja ta szuka pliku z nazwą tej biblioteki jako podstawą jego nazwy oraz odpowiedniego pliku źródłowego lub rozszerzenia natywnej biblioteki. Jeśli katalog zawiera zarówno plik źródłowy `.rb`, jak i plik rozszerzenia binarnego, funkcja `require` załadowuje plik źródłowy.
- Funkcja `load` może załadować jeden plik wiele razy. Funkcja `require` uniemożliwia wielokrotne załadowanie tego samego pliku (można ją jednak oszukać, używając dwóch różnych, ale równoważnych ścieżek do tego samego pliku bibliotecznego; w Ruby 1.9 funkcja ta rozwija względne ścieżki do ścieżek absolutnych, dzięki czemu trochę trudniej ją oszukać). Funkcja `require` pamięta, które pliki zostały załadowane, dodając je do globalnej tablicy `$"` (znanej także jako `$LOADED_FEATURES`). Funkcja `load` tego nie robi.
- Funkcja `load` ładuje wyznaczony plik na bieżącym poziomie `$SAFE`. Funkcja `require` ładuje wyznaczoną bibliotekę przy `$SAFE` ustalonym na 0, nawet jeśli kod wywoływany przez tę funkcję posiada wyższą wartość dla tej zmiennej. Więcej na temat zmiennej `$SAFE` i systemu zabezpieczeń Ruby znajduje się w podrozdziale 10.5 (należy zauważyć, że jeśli zmienna `$SAFE` ma wartość większą od 0, funkcja `require` nie załadowuje żadnego pliku z niepevną nazwą (ang. *tainted*) ani z katalogu z globalnymi uprawnieniami zapisu; teoretycznie zatem powinno być bezpieczne dla funkcji `require` ładowanie plików z obniżonym poziomem `$SAFE`).

Kolejne podrozdziały bardziej szczegółowo opisują działanie funkcji `load` i `require`.

7.6.1. Ścieżka wczytywania

Ścieżka wczytywania w Ruby to tablica, do której dostęp można uzyskać za pomocą dwóch zmiennych globalnych — `$LOAD_PATH` i `$:` (dwukropk jest używany jako znak rozdzielający ścieżki w systemach uniksowych). Każdy element tej tablicy jest nazwą katalogu, który zostanie przeszukany przez Ruby w celu znalezienia plików do załadowania. Najpierw przeszukiwane są katalogi z początku tablicy. W Ruby 1.8 elementy tablicy `$LOAD_PATH` muszą być łańcuchami. W Ruby 1.9 mogą to być łańcuchy lub dowolne obiekty udostępniające metodę `to_path` zwracającą łańcuch.

Domyślna wartość tablicy `$LOAD_PATH` jest uzależniona od implementacji Ruby, systemu operacyjnego, w którym działa, a nawet miejsca w systemie operacyjnym, gdzie Ruby został zainstalowany. Oto typowa wartość Ruby 1.8 uzyskana za pomocą polecenia `ruby -e 'puts $:'`:

```
/usr/lib/site_ruby/1.8  
/usr/lib/site_ruby/1.8/i386-linux  
/usr/lib/site_ruby
```

```
/usr/lib/ruby/1.8  
/usr/lib/ruby/1.8/i386-linux
```

Katalog `/usr/lib/ruby/1.8` to katalog, w którym zainstalowano bibliotekę standardową. Katalog `/usr/lib/site_ruby/1.8/i386-linux` przechowuje rozszerzania binarne biblioteki standardowej systemu Linux. Katalogi `site_ruby` przechowują biblioteki, które zostały zainstalowane w danej lokalizacji. Należy zauważyć, że katalogi te są przeszukiwane na początku, co oznacza, że bibliotekę standardową można przesłonić za pomocą plików umieszczonych w tym miejscu. Aktualny katalog roboczy „..” znajduje się na końcu ścieżki przeszukiwania. Jest to katalog, w którym użytkownik uruchomił program. Nie jest to jednak to samo co katalog, w którym program został zainstalowany.

W Ruby 1.9 domyślna ścieżka wczytywania jest bardziej skomplikowana. Oto typowa wartość:

```
/usr/local/lib/ruby/gems/1.9/gems/rake-0.7.3/lib  
/usr/local/lib/ruby/gems/1.9/gems/rake-0.7.3/bin  
/usr/local/lib/ruby/site_ruby/1.9  
/usr/local/lib/ruby/site_ruby/1.9/i686-linux  
/usr/local/lib/ruby/site_ruby  
/usr/local/lib/ruby/vendor_ruby/1.9  
/usr/local/lib/ruby/vendor_ruby/1.9/i686-linux  
/usr/local/lib/ruby/vendor_ruby  
/usr/local/lib/ruby/1.9  
/usr/local/lib/ruby/1.9/i686-linux
```

Jedną niewielką zmianą w ścieżce wczytywania w Ruby 1.9 jest dodanie katalogów `vendor_ruby`, które są przeszukiwane po `site_ruby` i przed biblioteką standardową. Są przeznaczone do przechowywania usprawnień dostarczonych przez dostawców systemów operacyjnych.

Jedną z ważniejszych zmian jest dodanie katalogów instalacyjnych dla gemów. W widocznej powyżej ścieżce, dwa pierwsze katalogi przechowują pakiet `rake` zainstalowany za pomocą polecenia `gem` systemu zarządzania pakietami *RubyGems*. W tym przykładzie jest zainstalowany tylko jeden gem, ale jeśli zostanie zainstalowanych dużo gemów, ścieżka ta może stać się bardzo długa (uruchamiając programy niewykorzystujące gemów, można zyskać nieco lepszą wydajność, wywołując interpreter z opcją `--disable-gems` — wyłącza ona katalogi gemów ze ścieżki przeszukiwania). Jeśli zainstalowanych jest kilka wersji jednego gema, w domyślnej ścieżce wczytywania uwzględniana jest wersja oznaczona największym numerem. Można to zmienić za pomocą metody `Kernel.gem`.

System RubyGems jest wbudowany w Ruby 1.9 — polecenie `gem` jest zawsze dostępne w Ruby i można za jego pomocą zainstalać nowe pakiety, których katalogi instalacyjne są automatycznie dodawane do domyślnej ścieżki wczytywania. W Ruby 1.8 narzędzie RubyGems trzeba zainstalować osobno (choć niektóre dystrybucje mogą mieć je dołączone automatycznie), a katalogi gemów nie są dodawane do ścieżki wczytywania. Natomiast programy Ruby 1.8 wymagają modułu `rubygems`. Jego dołączenie powoduje, że domyślna metoda `require` zostaje zastąpiona nową wersją, która wie, gdzie szukać zainstalowanych gemów. Więcej na temat narzędzia RubyGems znajduje się w podrozdziale 1.2.5.

Aby dodać nowe katalogi na początku ścieżki przeszukiwania, należy użyć opcji wiersza poleceń `-I`. Aby dodać kilka katalogów, należy użyć opcji `-I` kilkakrotnie. Można także użyć jednej opcji `-I`, a poszczególne katalogi oddzielić dwukropkami (lub średnikami w systemie Windows).

Programy Ruby mogą również modyfikować swoją ścieżkę wczytywania, zmieniając zawartość tablicy `$LOAD_PATH`. Na przykład:

```
# Usuwa bieżący katalog ze ścieżki wczytywania.  
$:.pop if $:.last == '.'  
# Dodaje katalog instalacyjny bieżącego programu na  
# początek ścieżki wczytywania jako alternatywa użycia require_relative.  
$LOAD_PATH.unshift File.expand_path(__FILE__)  
# Dodaje wartość zmiennej środowiskowej na końcu ścieżki.  
$LOAD_PATH << ENV['MY_LIBRARY_DIRECTORY']
```

Na koniec należy zapamiętać, że ścieżkę wczytywania można całkowicie obejść, przekazując do metod `load` i `require` bezwzględne nazwy plików (zaczynające się od znaku `/` lub `~`).

7.6.2. Wykonywanie wczytanego kodu

Funkcje `load` i `require` wykonują kod zawarty w wyznaczonych plikach natychmiast. Jednak wywołanie tych metod nie jest równoważne z zastąpieniem ich wywołań kodem znajdującym się w pliku³.

Pliki załadowane za pomocą metod `load` i `require` są wykonywane w nowym zakresie najwyższego poziomu innym od tego, w którym metody te zostały wywołane. Załadowany plik widzi wszystkie zmienne globalne i stałe, które były zdefiniowane w chwili jego wczytywania, ale nie ma dostępu do lokalnego zakresu, w którym zaczęło się wczytywanie. Implikacje tego są następujące:

- Zmienne lokalne zdefiniowane w zakresie, z którego została wywołana metoda `load` lub `require`, są niewidoczne dla załadowanego pliku.
- Wszystkie zmienne lokalne utworzone przez ładowany plik są usuwane po zakończeniu ładowania — nie są nigdy widoczne poza plikiem, w którym zostały zdefiniowane.
- Na początku wczytywanego pliku wartość `self` jest zawsze głównym obiektem, tak samo jak przy uruchamianiu interpretera Ruby. To znaczy że wywołanie metody `load` lub `require` w wywołaniu metody nie powoduje propagacji obiektu odbiorcy do załadowanego pliku.
- Bieżące zagnieźdżenie modułów jest w załadowanym pliku ignorowane. Nie można na przykład otworzyć klasy i załadować pliku z definicjami metod. Plik ten zostanie przetworzony w zakresie najwyższego rzędu, a nie wewnątrz klasy lub modułu.

7.6.2.1. Pakowanie ładowanych plików

Metoda `load` posiada jeszcze jedną rzadko używaną cechę, o której do tej pory nie wspominaliśmy. Jeśli zostanie wywołana z drugim argumentem o wartości innej niż `nil` i `false`, to „opakowuje” wyznaczony plik i ładuje go do anonimowego modułu. Oznacza to, że taki plik nie ma wpływu na globalną przestrzeń nazw — wszystkie stałe (w tym klasy i moduły) w nim zdefiniowane są uwiezione w tym anonimowym module. Techniki tej można używać jako środka ostrożności (lub sposobu na zminimalizowanie liczby błędów spowodowanych kolizjami przestrzeni nazw). W podrozdziale 10.5 dowiesz się, że kiedy Ruby wykonuje kod

³ Inne wyjaśnienie dla programistów C: metody `load` i `require` są inne niż dyrektywa `#include` w C. Bliższe bezpośredniemu wstawieniu kodu do pliku jest przekazanie załadowanego kodu do globalnej funkcji `eval`: `eval(File.read(filename))`. Ale nawet to nie jest tym samym, ponieważ funkcja `eval` nie ustawia zmiennych lokalnych.

z niezaufanego źródła w „piaskownicy”, kod ten nie może wywoływać metody `require`, a metodę `load` tylko, jeśli jest to pakowanie ładowanego kodu.

Plik załadowany do anonimowego modułu może ustawiać zmienne globalne i zmienne przez niego ustawiane są widoczne dla kodu, który go załadował. Założmy, że napisałeś plik o nazwie `util.rb` zawierający definicję modułu `Util` będącego zbiorem przydatnych metod użytkowych. Aby metody te były dostępne nawet wtedy, gdy plik zostanie opakowany po załadowaniu, na końcu tego pliku można dodać poniższy wiersz kodu:

```
$Util = Util    # Zapisanie referencji do tego modułu w zmiennej globalnej.
```

Warto zauważyć, że kod ładujący plik `util.rb` do anonimowej przestrzeni nazw może uzyskać dostęp do tych funkcji użytkowych poprzez globalną zmienną `$Util` zamiast stałej `Util`.

W Ruby 1.8 możliwe jest nawet przekazanie anonimowego modułu z powrotem do kodu ładującego:

```
if Module.nesting.size > 0          # Jeśli zostałeś załadowany do modułu opakowującego,  
  $wrapper = Module.nesting[0]      # przekazujesz ten moduł z powrotem do kodu ładującego.  
end
```

Więcej informacji na temat metody `Module.nesting` znajduje się w podrozdziale 8.1.1.

7.6.3. Automatyczne ładowanie modułów

Metody `autoload` z modułu `Kernel` i klasy `Module` pozwalają na opóźnione ładowanie plików — kiedy są potrzebne. Globalna funkcja `autoload` pozwala zarejestrować nazwę niezdefiniowanej stałej (zazwyczaj jest to nazwa klasy lub modułu) i nazwę biblioteki, która ją definiuje. Przy pierwszym odwołaniu do tej stałej wyznaczona biblioteka jest ładowana za pomocą metody `require`. Na przykład:

```
# Doda 'socket', jeśli i gdy stała TCPSocket zostanie użyta po raz pierwszy.  
autoload :TCPSocket, "socket"
```

Klasa `Module` zawiera własną definicję metody `autoload` działającą ze stałymi zagnieżdżonymi w innym module.

Aby sprawdzić, czy odwołanie do stałej spowoduje załadowanie pliku, należy użyć metody `autoload?` lub `Moduleautoload?`. Metoda ta przyjmuje argument w postaci symbolu. Jeśli odwołanie do stałej wskazywanej przez ten symbol spowoduje załadowanie pliku, metoda ta zwraca nazwę tego pliku. W przeciwnym przypadku (jeżeli nie ma żadnego żądania automatycznego załadowania pliku lub plik został już wczytany) metoda `autoload?` zwraca wartość `nil`.

7.7. Metody singletonowe i klasa `eigenclass`

W rozdziale 6. zostało napisane, że istnieje możliwość definiowania metod singletonowych — metod, które są zdefiniowane dla tylko jednego obiektu zamiast całej klasy obiektów. Definicja metody singletonowej `sum` dla obiektu klasy `Point` może wyglądać następująco:

```
def Point.sum  
  # Ciało metody.  
end
```

Jak było wspomniane wcześniej w tym rozdziale, metody klasowe klasy są właśnie metodami singletonowymi egzemplarza klasy `Class` reprezentującego tę klasę.

Metody singletonowe obiektu nie są definiowane przez klasę tego obiektu. Są jednak metodami, a więc muszą być związane z jakąś klasą. Metody singletonowe obiektu są metodami egzemplarza anonimowej klasy `eigenclass` związanej z tym obiektem. Niemieckie słowo „eigen” oznacza (mniej więcej) „własny”, „właściwy czemuś” lub „charakterystyczny dla”. Klasa `eigenclass` jest również nazywana **klasą singletonową** lub (rzadziej) **metaklasą**. Określenie „`eigenclass`” nie jest ogólnie przyjęte w społeczności Ruby, ale w tej książce będziemy się nim posługiwać.

W Ruby jest dostępna składnia pozwalająca otworzyć klasę `eigenclass` obiektu w celu dodania do niej metod. Jest to alternatywa dla definiowania metod singletonowych pojedynczo — można w zamian zdefiniować dowolną liczbę metod egzemplarza klasy `eigenclass`. Aby otworzyć klasę `eigenclass` obiektu `o`, należy napisać `class << o`. Na przykład poniżej zdefiniowane zostały metody klasowe obiektu klasowego `Point`:

```
class << Point
  def class_method1          # Metoda egzemplarza klasy eigenclass.
  end                        # Jest to także metoda klasowa obiektu klasy Point.
  def class_method2
  end
end
```

Jeśli klasa `eigenclass` obiektu klasy zostanie otwarta wewnątrz definicji tej klasy, zamiast powtarzać nazwę tej klasy, można użyć słowa `self`. Spójrz jeszcze raz na przykładowy kod prezentowany wcześniej w tym rozdziale:

```
class Point
  # Metody egzemplarza.
  class << self
    # Metody klasowe jako metody egzemplarza klasy eigenclass.
  end
end
```

Pisząc kod, należy uważać. Pomiędzy trzema poniższymi wierszami kodu jest znaczna różnica:

```
class Point           # Utworzenie lub otwarcie klasy Point.
class Point3D < Point # Utworzenie podklasy klasy Point.
class << Point       # Otwarcie klasy eigenclass obiektu klasowego Point.
```

Ogólnie rzecz biorąc, indywidualne definicje klas singletonowych są wyraźniejsze niż jawne otwieranie klasy `eigenclass`.

Kiedy zostaje otwarta klasa `eigenclass` obiektu, `self` odnosi się do obiektu klasy `eigenclass`. Zatem aby uzyskać klasę `eigenclass` obiektu `o`, należy napisać:

```
eigenclass = class << o; self; end
```

Można to formalnie zapisać jako metodę klasy `Object`, dzięki czemu można będzie pytać o klasę `eigenclass` każdego obiektu:

```
class Object
  def eigenclass
    class << self; self; end
  end
end
```

Przedstawiona wyżej funkcja użytkowa `eigenclass` jest potrzebna tylko w zaawansowanym metaprogramowaniu. Warto jednak zrozumieć, co to są klasy `eigenclass`, ponieważ można się

na nie natknąć w istniejącym kodzie, poza tym są one ważnym składnikiem algorytmu rozstrzygania nazw metod, który opisujemy w kolejnym podrozdziale.

7.8. Wyszukiwanie metod

Kiedy ewaluowana jest wartość wyrażenia wywołania metody, Ruby musi najpierw zgadnąć, którą metodę ma wywołać. Proces ten nazywa się **wyszukiwaniem metody** lub **rozstrzyganiem nazwy metody**. Dla wyrażenia wywołania metody `o.m` rozstrzyganie nazwy metody odbywa się w następujący sposób:

1. Sprawdzenie klasy eigenclass obiektu `o` w poszukiwaniu metod singletonowych o nazwie `m`.
2. Jeśli w klasie eigenclass nie zostanie znaleziona żadna metoda o nazwie `m`, zostaje przeszukana klasa obiektu `o` w celu znalezienia metody egzemplarza o nazwie `m`.
3. W przypadku gdy w klasie tej nie zostanie znaleziona żadna metoda `m`, Ruby przeszukuje metody egzemplarza wszystkich modułów dołączonych do klasy `o`. Jeżeli klasa dołącza więcej niż jeden moduł, moduły te są przeszukiwane w odwrotnej kolejności względem kolejności ich dołączania. To znaczy ostatni dodany moduł jest przeszukiwany na początku.
4. Jeśli w klasie obiektu `o` i jej modułach nie zostanie znaleziona żadna metoda `m`, przeszukiwana jest nadklasa tej klasy. Kroki 2. i 3. są powtarzane dla każdej klasy obecnej w hierarchii dziedziczenia.
5. W sytuacji gdy w wyniku tych czynności nie zostanie znaleziona żadna metoda `m`, wywoływana jest metoda `method_missing`. Aby znaleźć jej definicję, algorytm rozstrzygania nazwy metody zostaje uruchomiony od punktu 1. Domyślną implementację metody `method_missing` zawiera moduł `Kernel`, a więc to drugie podejście daje gwarancję powodzenia. Więcej informacji na temat metody `method_missing` znajduje się w podrozdziale 8.4.5.

Przeanalizuj działanie tego algorytmu na konkretnym przykładzie. Założmy, że masz poniższy fragment kodu:

```
message = "witaj"  
message.world
```

Chcesz wywołać metodę `o` nazwie `world` na rzecz egzemplarza klasy `String` "hello". Procedura rozstrzygania nazwy przebiega następująco:

1. Sprawdzenie klasy eigenclass w poszukiwaniu metod singletonowych. W tym przypadku nie ma żadnych.
2. Sprawdzenie klasy `String`. Nie ma w niej żadnej metody egzemplarza o nazwie `world`.
3. Przeszukanie modułów `Comparable` i `Enumerable` (tylko Ruby 1.8) klasy `String` w celu znalezienia metody egzemplarza o nazwie `world`. Żaden z tych modułów nie definiuje takiej metody.
4. Sprawdzenie klasy `Object`, czyli nadklasy klasy `String`. Klasa `Object` również nie definiuje metody o nazwie `world`.
5. Sprawdzenie modułu `Kernel` dołączonego do klasy `Object`. Tutaj metoda `world` również jest nieobecna. Algorytm przechodzi do szukania metody `method_missing`.

6. Szukanie metody `method_missing` we wszystkich wymienionych do tej pory miejscach — w klasie `eigenclass` obiektu klasy `String`, modułach `Comparable` i `Enumerable`, klasie `Object` i module `Kernel`. Pierwsza znaleziona definicja metody `method_missing` znajduje się w module `Kernel` i to ona zostanie wywołana. Metoda ta zgłasza wyjątek:

```
NoMethodError: undefined method 'world' for "witaj":String
```

Można odnieść wrażenie, że każde wywołanie metody wymaga przeprowadzenia przez Ruby wyczerpujących poszukiwań. Jednak w typowych implementacjach poszukiwania zakończone powodzeniem są zapisywane w pamięci podręcznej, dzięki czemu kolejne poszukiwania tej samej nazwy (jeśli w międzyczasie nie dodano nowych definicji metod) będą bardzo szybkie.

7.8.1. Wyszukiwanie metod klasowych

Algorytm rozstrzygania nazw metod dla metod klasowych jest taki sam jak dla metod egzemplarza, ale jest jeden haczyk. Zaczniemy od prostego przypadku, bez haczyka. Poniżej znajduje się klasa `C`, która nie definiuje żadnych własnych metod:

```
class C  
end
```

Pamiętaj, że po zdefiniowaniu takiej klasy, stała `C` odnosi się do obiektu będącego egzemplarzem klasy `Class`. Wszystkie metody klasowe, które zdefiniujesz, będą metodami singletonowymi tego obiektu `C`.

Po zdefiniowaniu klasy `C` prawdopodobnie napiszesz wyrażenie wywołania metody z użyciem metody `new` tej klasy:

```
c = C.new
```

Poszukując metody `new`, Ruby najpierw szuka metod singletonowych w klasie `eigenclass` obiektu `C`. Nasza klasa nie posiada żadnych metod klasowych, a więc nic w niej nie zostanie znalezione. Po przeszukaniu klasy `eigenclass` algorytm przeszukuje klasę obiektu `C`. Obiekt `C` należy do klasy `Class`, a więc Ruby przeszukuje metody klasy `Class`, wśród których znajduje metodę egzemplarza o nazwie `new`.

Tu nie ma pomyłki. Algorytm rozstrzygania nazw metod dla metody klasowej `C.new` zakończył działanie, znajdując metodę egzemplarza `Class.new`. Rozróżnienie pomiędzy metodami egzemplarza a metodami klasowymi jest przydatne w paradygmacie programowania obiektowego. Jednak w języku Ruby, w którym klasy są reprezentowane przez obiekty, rozróżnienie to jest nieco sztuczne. Każde wywołanie metody, czy to egzemplarza, czy klasowej, składa się z obiektu odbiorcy i nazwy metody. Algorytm rozstrzygania nazw metod znajduje odpowiednią definicję metody dla tego obiektu. Obiekt `C` jest egzemplarzem klasy `Class`, dlatego można oczywiście wywoływać metody egzemplarza klasy `Class` poprzez ten obiekt. Ponadto klasa `Class` dziedziczy metody egzemplarza klas `Module` i `Object` oraz modułu `Kernel`, dzięki czemu metody te są również dostępne jako metody obiektu `C`. Jedyny powód, dla którego nazywa się je metodami klasowymi, jest taki, że obiekt `C` jest akurat klasą.

Metoda klasowa `C.new` została znaleziona jako metoda egzemplarza klasy `Class`. Gdyby jednak nie została tam znaleziona, algorytm rozstrzygania nazw metod kontynuowałby działanie tak jak dla metody egzemplarza. Po nieudanej próbie znalezienia szukanej metody w klasie `Class` zostałyby przeszukane moduły (klasa `Class` nie dołącza żadnego), a następnie nadklasa `Module`. Następnie przeszukane zostałyby moduły klasy `Module` (nie ma żadnych) i w końcu nadklasa `Object` klasy `Module` oraz jej moduł `Kernel`.

Wspomniany na początku haczyk polega na tym, że metody klasowe są dziedziczone tak samo jak metody egzemplarza. Oto definicja metody klasowej `Integer.parse`, która użыта zostanie jako przykład:

```
def Integer.parse(text)
  text.to_i
end
```

Ponieważ klasa `Fixnum` jest podklassą klasy `Integer`, metodę tę można wywołać w poniższym wyrażeniu:

```
n = Fixnum.parse("1")
```

Z przedstawionego wcześniej opisu algorytmu rozstrzygania nazw metod wiadomo, że Ruby najpierw przeszukałby klasę `eigenclass` obiektu klasy `Fixnum` w celu znalezienia metod singletonowych. Następnie poszukałby metod egzemplarza klas `Class`, `Module`, `Object` i modułu `Kernel`. Gdzie w takim razie znajdzie metodę `parse?` Metoda klasowa klasy `Integer` jest tylko metodą singletonową obiektu `Integer`, co oznacza, że jest zdefiniowana przez klasę `eigenclass` obiektu `Integer`. A więc w jaki sposób ta klasa `eigenclass` obiektu `Integer` włącza się w algorytm rozstrzygania nazw metod?

Obiekty klas są wyjątkowe, ponieważ posiadają nadklasy. Klasa `eigenclass` obiektów klas również są wyjątkowe, ponieważ także posiadają nadklasy. Klasa `eigenclass` zwykłego obiektu istnieje samodzielnie i nie ma żadnej nadklasy. Klasa `eigenclass` `Fixnum` i `Integer` będziemy nazywać `Fixnum'` i `Integer'`.

Wiedząc o tym haczyku, można pełniej wyjaśnić zasadę działania algorytmu rozstrzygania nazw metod i powiedzieć, że kiedy Ruby szuka metod singletonowych w klasie `eigenclass` obiektu, przeszukuje także nadklasę (i wszystkich przodków) tej klasy `eigenclass`. W związku z tym, szukając metody klasowej klasy `Fixnum`, Ruby najpierw sprawdził metody singletonowe `Fixnum`, `Integer`, `Numeric` i `Object`, a następnie metody egzemplarza klas `Class`, `Module` i `Object` oraz modułu `Kernel`.

7.9. Wyszukiwanie stałych

Kiedy interpreter napotka referencję do stałej bez kwalifikatora, musi znaleźć jej odpowiednią definicję. Do tego celu wykorzystuje algorytm rozstrzygania nazw, podobnie jak w przypadku nazw metod. Jednak algorytm rozstrzygania nazw stałych działa inaczej niż dla nazw metod.

Najpierw Ruby próbuje znaleźć definicję stałej w zakresie leksykalnym jej odwołania. Oznacza to, że na początku definicji stałej szuka w klasie lub module zawierającym odwołanie do tej stałej. Jeśli jej tam nie znajdzie, sprawdza kolejną otaczającą klasę lub moduł. Proces ten trwa do wyczerpania coraz szerzej otaczających klas i modułów. Należy zauważyć, że stałe najwyższego poziomu lub „globalne” nie są uznawane za część zakresu leksykalnego i nie są brane pod uwagę w tej części wyszukiwania definicji. Metoda klasowa `Module.nesting` zwraca listę klas i modułów, które są przeszukiwane na tym etapie w takiej kolejności, w jakiej są przeszukiwane.

W przypadku gdy w zakresie leksykalnym nie zostanie znaleziona żadna definicja stałej, Ruby próbuje znaleźć ją w hierarchii dziedziczenia, sprawdzając przodków klasy lub modułu, który zawiera odwołanie do tej stałej. Metoda `ancestors` nadrzędnej klasy lub modułu zwraca listę klas i modułów przeszukiwanych na tym etapie.

Jeżeli w hierarchii dziedziczenia nie zostanie znaleziona żadna definicja stałej, sprawdzane są definicje stałych najwyższego poziomu.

W sytuacji gdy nadal nie zostanie znaleziona szukana definicja stałej, zostaje wywołana metoda `const_missing` (jeśli istnieje) nadrzędnej klasy lub modułu. Może ona dostarczyć wartości dla tej stałej. Metoda `const_missing` została opisana w rozdziale 8., a sposób jej użycia ilustruje listing 8.3.

Kilka szczegółów dotyczących działania algorytmu rozstrzygania nazw stałych zasługuje na bardziej szczegółowe omówienie:

- Stałe zdefiniowane w modułach nadrzędnych są preferowane względem stałych zdefiniowanych w modułach dołączanych.
- Moduły dołączone przez klasę są przeszukiwane przed nadklassą tej klasy.
- Klasa `Object` jest częścią hierarchii dziedziczenia wszystkich klas. Stałe najwyższego poziomu, zdefiniowane poza wszystkimi klasami i modułami, są jak metody najwyższego poziomu — niejawnie należą do klasy `Object`. W związku z tym, kiedy w klasie znajduje się odwołanie do stałej najwyższego poziomu, jej definicja jest znajdowana podczas przeszukiwania hierarchii dziedziczenia. Jeśli natomiast odniesienie do takiej stałej znajduje się w definicji modułu, po przeszukaniu przodków tego modułu konieczne jest jawne sprawdzenie klasy `Object`.
- Moduł `Kernel` jest przodkiem klasy `Object`. Oznacza to, że stałe zdefiniowane w module `Kernel` zachowują się podobnie jak stałe najwyższego poziomu, ale mogą zostać przesłonięte przez prawdziwe stałe najwyższego poziomu zdefiniowane w klasie `Object`.

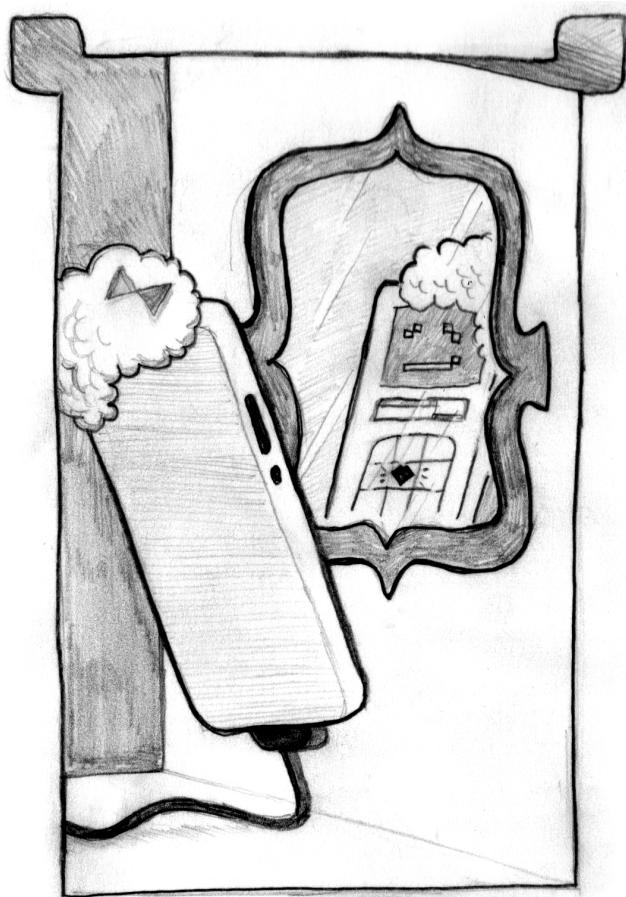
Listing 7.1 definiuje i odnajduje stałe w sześciu różnych zakresach oraz demonstruje opisany wcześniej algorytm rozstrzygania nazw stałych.

Listing 7.1. Rozstrzyganie nazw stałych

```
module Kernel
  # Stałe zdefiniowane w module Kernel.
  A = B = C = D = E = F = "zdefiniowane w module Kernel"
end
# Stałe globalne zdefiniowane w klasie Object.
A = B = C = D = E = "zdefiniowane na najwyższym poziomie"
class Super
  # Stałe zdefiniowane w nadklassie.
  A = B = C = D = "zdefiniowane w nadklassie"
end
module Included
  # Stałe zdefiniowane w dołączonym module.
  A = B = C = "zdefiniowane w dołączonym module"
end
module Enclosing
  # Stałe zdefiniowane w module nadrzędnym.
  A = B = "zdefiniowane w module nadrzędnym"
  class Local < Super
    include Included
    # Stała zdefiniowana lokalnie.
    A = "zdefiniowana lokalnie"
    # Lista przeszukiwanych modułów w kolejności przeszukiwania.
    #[Enclosing::Local, Enclosing, Included, Super, Object, Kernel].
    search = (Module.nesting + self.ancestors + Object.ancestors).uniq
    puts A # Drukuję „zdefiniowane lokalnie”.
    puts B # Drukuję „zdefiniowane w module nadrzędnym”.
    puts C # Drukuję „zdefiniowane w dołączonym module”.
  end
end
```

```
    puts D  # Drukuj „zdefiniowane w nadklasie”.
    puts E  # Drukuj „zdefiniowane na najwyższym poziomie”.
    puts F  # Drukuj „zdefiniowane w module Kernel”.
  end
end
```

Refleksja i metaprogramowanie



Przekonałeś się, że Ruby jest bardzo dynamicznym językiem programowania. Można wstawić do klas nowe metody w czasie działania programu, tworzyć aliasy istniejących metod, a nawet definiować metody dla pojedynczych obiektów. Dodatkowo język ten posiada bogate API refleksji. Refleksja, zwana także **introspekcją**, oznacza po prostu zdolność programu do badania swojego własnego stanu i struktury. Program w języku Ruby może na przykład zdobyć listę metod zdefiniowanych przez klasę Hash, sprawdzić wartość wybranej zmiennej egzemplarza w wyznaczonym obiekcie lub przejść iteracyjnie przez wszystkie obiekty klasy Regexp aktualnie zdefiniowane przez interpreter. API refleksji pozwala nawet na coś więcej, umożliwiając programowi modyfikowanie swojego własnego stanu i struktury. Program Ruby może dynamicznie ustawać wartości wyznaczonych zmiennych, wywoływać wyznaczone metody, a nawet definiować nowe klasy i metody.

API refleksyjne języka Ruby — wraz z jego dynamiczną naturą, strukturami sterującymi opartymi na blokach i iteratorach oraz składnią pozwalającą na opuszczanie nawiasów — sprawia, że język ten idealnie nadaje się do **metaprogramowania**. Mówiąc krótko, metaprogramowanie to pisanie programów (lub frameworków), które pomagają pisać programy. Innymi słowy, jest to zestaw technik pozwalających rozszerzyć składnię języka Ruby w taki sposób, aby ułatwić sobie programowanie. Metaprogramowanie jest ściśle związane z technologią pisania **języków do wyspecjalizowanych zastosowań** (ang. *Domain Specific Languages* — DSL). Języki DSL w Ruby zazwyczaj używają wywołań metod i bloków, jakby były słowami kluczowymi w rozszerzeniach języka stworzonych do rozwiązywania konkretnych zadań.

Niniejszy rozdział zaczyna się od kilku podrozdziałów wprowadzających do API refleksyjnego języka Ruby. API to jest zaskakująco bogate i zawiera całkiem sporą liczbę metod. Większość z nich jest zdefiniowana w module Kernel i klasach Object i Module.

Czytając te wstępne podrozdziały, należy pamiętać, że refleksja nie jest sama w sobie metaprogramowaniem. Metaprogramowanie zazwyczaj w jakiś sposób rozszerza składnię lub zakres działania języka Ruby i zwykle jest związane z więcej niż jednym rodzajem refleksji. Po wprowadzeniu do podstawowego API refleksyjnego kolejne podrozdziały przechodzą do demonstracji na przykładach najczęściej używanych technik metaprogramowania z wykorzystaniem tego API.

Rozdział ten opisuje zaawansowaną tematykę. Można być dobrym programistą, nigdy go nie czytając. Pomoże może być przeczytanie najpierw pozostałych rozdziałów tej książki, a następnie wrócenie do tego. Rozdział ten można potraktować jako pewnego rodzaju egzamin — jeśli rozumiesz zaprezentowane tu przykładowe programy (zwłaszcza te dłuższe z końca), opanowałeś język programowania Ruby!

8.1. Typy, klasy i moduły

Najczęściej używane metody refleksyjne to te, które pozwalają określić typ obiektu — jakiej klasy jest egzemplarzem i jakie metody można na jego rzecz wywoływać. Większość z tych ważnych metod została wprowadzona wcześniej w rozdziale 3.8.4. Przypomnijmy:

`o.class`

Zwraca klasę obiektu o.

`c.superclass`

Zwraca nadklasę klasy c.

`o.instance_of? c`
 Sprawdza, czy obiekt `o.class == c`.

`o.is_a? c`
 Sprawdza, czy `o` jest egzemplarzem klasy `c` lub jednej z jej podklas. Jeśli `c` jest modelem, metoda ta sprawdza, czy klasa obiektu `o.class` (lub którykolwiek z jej przodków) dołącza ten moduł.

`o.kind_of? c`
 Nazwa `kind_of?` jest synonimem nazwy `is_a?`.

`c === o`
 Dla dowolnej klasy lub modułu `c` sprawdza, czy `o.is_a?(c)`.

`o.respond_to? nazwa`
 Sprawdza, czy obiekt `o` udostępnia publiczną lub chronioną metodę `o` podanej nazwie. Przekazanie wartości `true` jako drugiego argumentu powoduje, że metoda ta sprawdza także metody prywatne.

8.1.1. Przodkowie i moduły

Poza metodami, które już znasz, istnieje jeszcze kilka związanych z nimi metod refleksyjnych służących do sprawdzania przodków klas i modułów oraz tego, które moduły dana klasa lub moduł dołącza. Działanie tych metod najłatwiej zrozumieć na przykładzie:

```
module A; end          # Pusty moduł.
module B; include A; end; # Moduł B dołącza moduł A.
class C; include B; end; # Klasa C dołącza moduł B.
C < B                 # => true — C dołącza B.
B < A                 # => true — B dołącza A.
C < A                 # => true.
Fixnum < Integer      # => true — wszystkie obiekty klasy Fixnum należą do klasy Integer.
Integer < Comparable   # => true — liczby całkowite można porównywać.
Integer < Fixnum       # => false — nie wszystkie obiekty klasy Integer należą do klasy Fixnum.
String < Numeric        # => nil —łańcuchy nie są liczbami.
A.ancestors            # => [A].
B.ancestors            # => [B, A].
C.ancestors            # => [C, B, A, Object, Kernel].
String.ancestors       # => [String, Enumerable, Comparable, Object, Kernel].
# Uwaga: w Ruby 1.9 klasa String nie jest już przeliczalna.
C.include?(B)          # => true.
C.include?(A)          # => true.
B.include?(A)          # => true.
A.include?(A)          # => false.
A.include?(B)          # => false.
A.included_modules     # => [].
B.included_modules     # => [A].
C.included_modules     # => [B, A, Kernel].
```

Niniejszy kod demonstruje użycie publicznej metody egzemplarza klasy `Module` o nazwie `include?`. Można też znaleźć dwa wywołania metody `include` (bez znaku zapytania), która jest prywatną metodą egzemplarza w klasie `Module`. Jako metoda prywatna może być tylko wywoływana niejawnie na rzecz obiektu `self`, co ogranicza zakres jej użycia do ciała definicji klasy lub modułu. Taki sposób użycia metody `include`, jakby była słowem kluczowym, jest przykładem metaprogramowania w rdzennej składni języka Ruby.

Metodą związaną z prywatną metodą `include` jest publiczna metoda `Object.extend`. Rozszerza ona obiekt, zamieniając metody egzemplarza każdego z wyznaczonych modułów w metody singletonowe tego obiektu:

```
module Greeter; def hi; "witaj"; end; end # Mało przydatny moduł.  
s = "obiekt klasy String"  
s.extend(Greeter)      # Dodanie hi jako metody singletonowej do obiektu s.  
s.hi                  # => "witaj".  
String.extend(Greeter) # Dodanie hi jako metody klasowej klasy String.  
String.hi              # => "witaj".
```

Metoda klasowa `Module.nesting` nie jest związana z dołączaniem modułów lub dziedziczeniem. Zwraca ona tablicę zawierającą informacje o zagnieżdżaniu modułów w bieżącej lokalizacji. Element `Module.nesting[0]` to bieżąca klasa lub moduł, `Module.nesting[1]` to nadziedziona klasa lub moduł itd.:

```
module M  
  class C  
    Module.nesting    # => [M::C, M]  
  end  
end
```

8.1.2. Definiowanie klas i modułów

Klasy i moduły są egzemplarzami klas `Class` i `Module`. Jako takie można je tworzyć dynamicznie:

```
M = Module.new      # Definicja nowego modułu o nazwie M.  
C = Class.new       # Definicja nowej klasy o nazwie C.  
D = Class.new(C) { # Definicja podklasy klasy C,  
  include M        # która dolicza moduł M.  
}  
D.to_s             # => D — klasa zdobywa nazwę stałej w czarodziejski sposób.
```

Jedną z przyjemnych własności języka Ruby jest to, że kiedy dynamicznie utworzony匿名owy moduł lub klasa zostanie przypisana do stałej, nazwa tej stałej staje się nazwą tego modułu lub tej klasy (i jest zwracana przez metody `name` i `to_s`).

8.2. Wykonywanie łańcuchów i bloków

Jedną z najpotężniejszych i najprostszych własności refleksyjnych języka Ruby jest metoda `eval`. Jeśli program jest w stanie wygenerować łańcuch poprawnego kodu Ruby, metoda `Kernel.eval` może obliczyć wartość tego kodu:

```
x = 1  
eval "x + 1" # => 2.
```

Metoda `eval` daje bardzo duże możliwości, ale rzadko jest potrzebna w innych sytuacjach niż podczas pisania programów powłoki (jak `irb`) wykonujących wiersze kodu Ruby wprowadzane przez użytkownika (w programowaniu sieciowym wywoływanie metody `eval` jest prawie zawsze niebezpieczne, ponieważ kod odebrany od użytkownika może być złośliwy). Niedoświadczeni programiści czasami używają tej metody jako podpory w swoich programach. Jeśli używasz tej metody w swoim kodzie, sprawdź, czy nie można jej zastąpić czymś innym. Po tym wstępnie trzeba dodać, że jest kilka bardziej przydatnych sposobów na wykorzystanie metody `eval` i metod do niej podobnych.

8.2.1. Dowiązania i metoda eval

Obiekt klasy `Binding` reprezentuje stan dowiązań zmiennej w określonym momencie. Obiekt `Kernel.binding` zwraca dowiązania efektywne w miejscu wywołania. Obiekt klasy `Binding` można przekazać do metody `eval` jako drugi argument, a podany łańcuch zostanie wykonany w kontekście tych dowiązań. Jeśli na przykład zdefiniujesz metodę egzemplarza zwracającą obiekt klasy `Binding` reprezentujący dowiązania zmiennych wewnętrz jakiegoś obiektu, możesz wykorzystać te dowiązania do sprawdzenia i ustawienia zmiennych egzemplarza tego obiektu. Można to wykonać w następujący sposób:

```
class Object          # Otwarcie klasy Object, aby dodać nową metodę.
  def bindings      # Nazwa tej metody ma formę liczby mnogiej.
    binding         # To jest predefiniowana metoda modułu Kernel.
  end
end
class Test           # Prosta klasa ze zmienną egzemplarza.
  def initialize(x); @x = x; end
end
t = Test.new(10)      # Utworzenie obiektu klasy Test.
eval("@x", t.bindings) #=> 10 — zajrzałeś do środka obiektu t.
```

Definiowanie tego typu metody `Object.bindings`, aby zajrzeć do zmiennych egzemplarza w obiekcie, nie jest konieczne. Istnieje kilka metod (zostały opisane nieco dalej), które pozwalają w łatwiejszy sposób sprawdzić (i ustawić) wartość zmiennej egzemplarza w obiekcie.

Jak napisaliśmy w podrozdziale 6.6.2, obiekt klasy `Proc` definiuje publiczną metodę o nazwie `binding`, która zwraca obiekt klasy `Binding` reprezentujący dowiązania zmiennych efektywne dla ciała tego obiektu klasy `Proc`. Ponadto metoda `eval` przyjmuje także obiekt klasy `Proc` zamiast obiektu klasy `Binding` jako drugi argument.

Ruby 1.9 definiuje metodę `eval` na obiektach klasy `Binding`, dzięki czemu, zamiast przekazywać obiekt klasy `Binding` jako drugi argument do globalnej metody `eval`, można wywołać metodę `eval` na rzecz obiektu klasy `Binding`. Wybór jednego z tych dwóch sposobów jest kwestią stylu. Obie techniki są sobie równoważne.

8.2.2. Metody `instance_eval` i `class_eval`

W klasie `Object` jest zdefiniowana metoda o nazwie `instance_eval`, a w klasie `Module` metoda o nazwie `class_eval` (`module_eval` jest synonimem metody `class_eval`). Obie metody wykonują kod Ruby tak jak metoda `eval`, ale są dwie ważne różnice. Pierwsza polega na tym, że metody te wykonują kod w kontekście wyznaczonego obiektu lub w kontekście wyznaczonego modułu — obiekt ten lub moduł jest wartością `self`, podczas gdy kod jest wykonywany. Oto kilka przykładów:

```
o.instance_eval("@x") # Zwraca wartość zmiennej egzemplarza @x obiektu o.
# Definiuje metodę egzemplarza o nazwie len klasy String zwracającą długość łańcucha.
String.class_eval("def len; size; end")
# Oto inny sposób na zrobienie tego.
String.class_eval("alias len size")
# Definicja metody klasowej String.empty za pomocą metody instance_eval.
# Zwrócić uwagę, że cudzysłówki w cudzysłowach robią się niebezpieczne...
String.instance_eval("def empty; ''; end")
```

Zwróci uwagę na subtelną, ale o kluczowym znaczeniu różnicę pomiędzy metodami `instance_eval` a `class_eval`, kiedy wykonywany kod zawiera definicję metody. Metoda `instance_eval`

definiuje metody singletonowe obiektu (co powoduje powstanie metod klasowych, jeśli zostanie wywołana na rzecz obiektu klasy). Metoda `class_eval` definiuje zwykłe metody egzemplarza.

Druga ważna różnica między tymi dwiema metodami a globalną metodą `eval` jest taka, że `instance_eval` i `class_eval` mogą przyjmować bloki kodu do wykonania. Kiedy zamiast łańcucha zostanie przekazany blok kodu, zostanie on wykonany w odpowiednim kontekście. Poniżej przedstawione są alternatywne wersje zademonstrowanych wcześniej wywołań:

```
o.instance_eval { @x }
String.class_eval {
  def len
    size
  end
}
String.class_eval { alias len size }
String.instance_eval { def empty; ""; end }
```

8.2.3. Metody `instance_exec` i `class_exec`

W Ruby 1.9 dostępne są jeszcze dwie metody wykonujące kod — `instance_exec` i `class_exec` (oraz jej alias `module_exec`). Metody te wykonują blok (nie łańcuch) kodu w kontekście obiektu odbiorcy, tak jak to robią metody `instance_eval` i `class_eval`. Różnica polega na tym, że metody `exec` przyjmują argumenty, które przekazują do bloku. Dzięki temu blok kodu jest wykonywany w kontekście wyznaczonego obiektu z parametrami, których wartości pochodzą spoza tego obiektu.

8.3. Zmienne i stałe

Moduł `Kernel` i klasy `Object` i `Module` udostępniają metody refleksywne zwracające nazwy (w postaci łańcuchów) wszystkich zdefiniowanych zmiennych globalnych, aktualnie zdefiniowanych zmiennych lokalnych, wszystkich zmiennych egzemplarza w obiekcie, wszystkich zmiennych klasy lub modułu i wszystkich stałych klasy lub modułu:

```
global_variables #=> ["$DEBUG", "$SAFE", ...].
x = 1           # Definicja zmiennej lokalnej.
local_variables #=> ["x"].
# Definicja prostej klasy.
class Point
  def initialize(x,y); @x,@y = x,y; end # Definicja zmiennych egzemplarza.
  @@classvar = 1                      # Definicja zmiennej klasowej.
  ORIGIN = Point.new(0,0)              # Definicja stałej.
end
Point::ORIGIN.instance_variables #=> ["@y", "@x"].
Point.class_variables          #=> ["@@classvar"].
Point.constants                  #=> ["ORIGIN"].
```

Metody `global_variables`, `local_variables`, `instance_variables`, `class_variables` i `constants` w Ruby 1.8 zwracają tablice łańcuchów, a w Ruby 1.9 tablice symboli.

8.3.1. Sprawdzanie, ustawianie i testowanie zmiennych

Poza zwracaniem zdefiniowanych zmiennych i stałych klasy `Object` i `Module` udostępniają także metody refleksywne służące do sprawdzania, ustawiania i usuwania zmiennych egzemplarza, zmiennych klasowych i stałych. Nie istnieją żadne wyspecjalizowane metody do spraw-

dzania i ustawiania wartości zmiennych lokalnych i globalnych, ale do tego celu można użyć metody eval:

```
x = 1
varname = "x"
eval(varname)          #=> 1.
eval("varname = '$g'") # Ustawia varname na "$g".
eval("#{varname} = x") # Ustawia $g na 1.
eval(varname)          #=> 1.
```

Zauważ, że metoda eval wykonuje swój kod w zakresie tymczasowym. Metoda ta może zmienić wartość zmiennych egzemplarza, które istnieją. Ale wszystkie nowe zmienne egzemplarza definiowane przez tę metodę są lokalne w jej wywołaniu i przestają istnieć po zwróceniu przez nią wartości (to tak, jakby wykonywany blok kodu działał w ciele bloku — zmienne lokalne w bloku nie istnieją na zewnątrz tego bloku).

Można sprawdzać wartości, ustawać wartości i sprawdzać istnienie zmiennych egzemplarza dowolnego obiektu oraz zmiennych klasowych i stałych w dowolnej klasie i dowolnym module:

```
o = Object.new
o.instance_variable_set(:@x, 0)    # Zauważ wymagany prefiks @.
o.instance_variable_get(:@x)        #=> 0.
o.instance_variable_defined?(:@x)  #=> true.
Object.class_variable_set(:@@x, 1)   # Prywatna w Ruby 1.8.
Object.class_variable_get(:@@x)     # Prywatna w Ruby 1.8.
Object.class_variable_defined?(:@@x) #=> true; Ruby 1.9 i późniejsze wersje.
Math.const_set(:EPI, Math::E*Math::PI)
Math.const_get(:EPI)               #=> 8.53973422267357.
Math.const_defined? :EPI           #=> true.
```

W Ruby 1.9 jako drugi argument do metody const_get i const_defined? można przekazać wartość false, aby zaznaczyć, że interesuje nas tylko bieżąca klasa lub bieżący moduł i nie interesują nas odziedziczone stałe.

Metody służące do sprawdzania i ustawiania zmiennych klasowych są w Ruby 1.8 prywatne. W tej wersji języka można je wywoływać za pomocą metody class_eval:

```
String.class_eval { class_variable_set(:@@x, 1) } # Ustawienie zmiennej @@x w klasie String.
String.class_eval { class_variable_get(:@@x) }      #=> 1.
```

Klasy Object i Module udostępniają metody prywatne pozwalające oddefiniować zmienne egzemplarza, zmienne klasowe i stałe. Wszystkie one zwracają wartość usuniętej zmiennej lub stałej. Ponieważ są to metody prywatne, nie można wywoływać ich bezpośrednio na rzecz obiektów, klas ani modułów. Konieczne jest posłużenie się metodą eval lub send (zostały opisane dalej w tym rozdziale):

```
o.instance_eval { remove_instance_variable :@x }
String.class_eval { remove_class_variable(:@@x) }
Math.send :remove_const, :EPI # Wywołanie metody prywatnej za pomocą metody send.
```

Gdy zostanie utworzona referencja do niezdefiniowanej stałej, wywoływana jest metoda const_missing, jeśli istnieje. Można tę metodę zdefiniować w taki sposób, aby zwracała wartość wyznaczonej stałej (funkcjonalność tę można na przykład wykorzystać do implementacji funkcji automatycznego ładowania klasy lub modułu na żądanie). Oto prostszy przykład:

```
def Symbol.const_missing(name)
  name # Zwrócenie nazwy stałej jako symbolu.
end
Symbol::Test #=> :Test — niezdefiniowana stała zostaje efektywnie skonwertowana na symbol.
```

8.4. Metody

Klasy Object i Module udostępniają kilka metod służących do sporządzania list metod, ich sprawdzania, wywoływania i definiowania. Każdą z tych kategorii zajmiemy się osobno.

8.4.1. Sporządzanie list metod

Klasa Object udostępnia metody pozwalające sporządzić listę nazw metod zdefiniowanych dla wybranego obiektu. Metody te zwracają tablice zawierające nazwy metod. Nazwy te w języku Ruby 1.8 są łańcuchami, a w Ruby 1.9 symbolami:

```
o = "łańcuch"
o.methods                      #=> [ nazwy wszystkich metod publicznych ].  
o.public_methods                #=> to samo.  
o.public_methods(false)         # Wyłączenie metod odziedziczonych.  
o.protected_methods             #=> [] — nie ma żadnych.  
o.private_methods               #=> Tablica wszystkich metod prywatnych.  
o.private_methods(false)        # Wyłączenie odziedziczonych metod prywatnych.  
def o.single; 1; end             # Definicja metody singletonowej.  
o.singleton_methods              #=> ["single"] (lub [:single] w 1.9).
```

Możliwe jest także uzyskanie danych na temat metod od samej klasy zamiast jej egzemplarza. Poniższe metody są zdefiniowane w klasie Module. Podobnie jak metody klasy Object, zwracają tablice łańcuchów w Ruby 1.8 i tablice symboli w Ruby 1.9:

```
String.instance_methods == "s".public_methods           #=> true.  
String.instance_methods(false) == "s".public_methods(false) #=> true.  
String.public_instance_methods == String.instance_methods #=> true.  
String.protected_instance_methods #=> [].  
String.private_instance_methods(false) #=> ["initialize_copy",  
#      "initialize"].
```

Przypomnijmy, że metody klasowe klas i modułów są metodami singletonowymi obiektów klasy Class lub Module. W związku z tym, aby uzyskać listę metod, należy użyć metody Object. `→singleton_methods`:

```
Math.singleton_methods #=> ["acos", "log10", "atan2", ... ].
```

Poza powyższymi metodami tworzącymi zestawienia klasa Module zawiera także kilka predykatów pozwalających sprawdzić, czy wybrana klasa lub moduł definiuje określoną metodę egzemplarza:

```
String.public_method_defined? :reverse    #=> true.  
String.protected_method_defined? :reverse   #=> false.  
String.private_method_defined? :initialize #=> true.  
String.method_defined? :upcase!            #=> true.
```

Metoda `Module.method_defined?` sprawdza, czy wskazana metoda jest zdefiniowana jako metoda publiczna lub chroniona. W zasadzie pełni taką samą funkcję jak metoda Object. `→respond_to?`. W Ruby 1.9 przekazanie jako drugi argument wartości `false` powoduje wyłączenie metod odziedziczonych.

8.4.2. Tworzenie obiektów klasy Method

Aby sprawdzić określoną metodę, należy wywołać metodę `method` na rzecz dowolnego obiektu lub metodę `instance_method` na rzecz dowolnego modułu. Pierwsza z nich zwraca związany z odbiorcą obiekt klasy `Method`, który można wywoływać, a druga obiekt klasy `UnboundMethod`.

W Ruby 1.9 można ograniczyć wyszukiwanie do metod publicznych, wywołując metody `public_method` i `public_instance_method`. Metody te i zwracane przez nie obiekty opisaliśmy w podrozdziale 6.7:

```
"s".method(:reverse)           # => Obiekt klasy Method.  
String.instance_method(:reverse) # => Obiekt klasy UnboundMethod.
```

8.4.3. Wywoływanie metod

Jak pisaliśmy wcześniej w tym rozdziale i podrozdziale 6.7, za pomocą metody `method` dowolnego obiektu można utworzyć obiekt klasy `Method` reprezentujący wyznaczoną metodę tego obiektu. Obiekty klasy `Method` udostępniają metodę `call`, podobnie jak obiekty klasy `Proc`. Za jej pomocą można wywołać tę metodę.

Zazwyczaj prościej jest wywołać wyznaczoną metodę określonego obiektu za pomocą metody `send`:

```
"witaj".send :upcase          # => "WITAJ": wywołanie metody egzemplarza.  
Math.send(:sin, Math::PI/2)    # => 1.0: wywołanie metody klasowej.
```

Metoda `send` wywołuje na rzecz swojego odbiorcy metodę, której nazwę podano jako jej pierwszy argument, wszystkie pozostałe argumenty przekazując do tej metody. Nazwa `send` (wyślij) pochodzi ze stylu programowania obiektowego, w którym wywoływanie metody jest nazywane „wysyłaniem komunikatu” do obiektu.

Metoda `send` może wywołać dowolną metodę dowolnego obiektu, w tym metody prywatne i chronione. Wcześniej przedstawiony był przykład wywołania za pomocą metody `send` prywatnej metody `remove_const` obiektu klasy `Module`. Ponieważ funkcje globalne są w rzeczywistości metodami prywatnymi klasy `Object`, za pomocą metody `send` można wywoływać te metody na rzecz dowolnych obiektów (chociaż nie jest to to, co chciałbyś robić):

```
"witaj".send :puts, "świecie"      # Drukuję "świecie".
```

W Ruby 1.9 dostępna jest alternatywa dla metody `send` o nazwie `public_send`. Działa ona tak samo jak metoda `send`, ale wywołuje tylko metody publiczne:

```
"witaj".public_send :puts, "świecie" # Powoduje wyjątek NoMethodError.
```

Metoda `send` jest podstawową metodą klasy `Object`, ale ma popularną nazwę i może zostać przesłonięta w podklasach. Dlatego w Ruby znajduje się jej synonim o nazwie `_send_`. Próba przesłonięcia lub oddefiniowania tego synonimu powoduje zgłoszenie ostrzeżenia.

8.4.4. Definiowanie, usuwanie definicji i tworzenie aliasów metod

Aby zdefiniować nową metodę egzemplarza klasy lub modułu, należy użyć metody `define_method`. Jest to metoda egzemplarza klasy `Module` przyjmująca jako pierwszy argument nazwę nowej metody (w postaci symbolu). Ciało metody dostarczane jest w postaci obiektu klasy `Method` przekazanego jako drugi argument lub w bloku. Ważne jest, aby zapamiętać, że metoda `define_method` jest prywatna. Aby ją wywołać, trzeba znajdować się w klasie lub module, na rzecz którego ma ona zostać użyta:

```
# Dodanie metody egzemplarza o nazwie m do klasy c z ciałem b.  
def add_method(c, m, &b)  
  c.class_eval {  
    define_method(m, &b)  
  }  
end
```

```
end
add_method(String, :greet) { "Witaj, " + self }
"świecie".greet #=> "Witaj, świecie".
```

Definiowanie atrybutowych metod dostępu

Metody attr_reader i attr_accessor (zobacz podrozdział 7.1.5) również definiują nowe metody w klasach. Podobnie jak metoda define_method są one prywatnymi metodami klasy Module i można je łatwo zaimplementować na podstawie metody define_method. Te metody służące do tworzenia metod są doskonałym przykładem przydatności metody define_method. Zauważ, że dzięki temu, iż metody te są przeznaczone do użytku wewnętrz klasy, nie przeszukadza im to, że metoda define_method jest prywatna.

Aby zdefiniować metodę klasową (lub dowolną metodę singletonową) za pomocą metody define_method, należy wywołać ją na rzecz klasy eigenklass:

```
def add_class_method(c, m, &b)
  eigenclass = class << c; self; end
  eigenclass.class_eval {
    define_method(m, &b)
  }
end
add_class_method(String, :greet) { |name| "Witaj, " + name }
String.greet("świecie") #=> "Witaj, świecie".
```

W Ruby 1.9 jest łatwiej, ponieważ można użyć metody define_singleton_method z klasy Object:

```
String.define_singleton_method(:greet) { |name| "Witaj, " + name }
```

Jedyną wadą metody define_method jest to, że nie pozwala ona na wyznaczenie ciała metody, która przyjmuje blok. Aby dynamicznie utworzyć metodę przyjmującą blok, należy użyć instrukcji def w połączeniu z metodą class_eval. Jeśli tworzona metoda jest wystarczająco dynamiczna, może być niemożliwe przekazanie bloku do metody class_eval i może okazać się konieczne zapisanie definicji tej metody jako łańcucha do wykonania. Przykłady tego przedstawione zostaną w dalszej części niniejszego rozdziału.

Aby utworzyć synonim lub alias istniejącej metody, można użyć zwykłej instrukcji alias:

```
alias plus +      # Słowo plus jest synonimem operatora +.
```

Czasami jednak w programowaniu dynamicznym konieczne jest użycie prywatnej metody klasy Module alias_method. Jako metoda może ona przyjąć jako argumenty dwa dowolne wyrażenia zamiast dwóch identyfikatorów wpisanych na stałe w kodzie źródłowym (jako metoda wymaga również przecinka między swoimi argumentami). Metoda alias_method jest często używana do **tworzenia łańcuchów aliasów** istniejących metod. Poniżej znajduje się prosty przykład (dalej jest ich więcej):

```
# Utworzenie aliasu dla metody m w klasie (lub module) c.
def backup(c, m, prefix="original")
  n = "#{prefix}_#{m}"      # Wyznaczenie aliasu.
  c.class_eval {            # Ponieważ metoda alias_method jest prywatna.
    alias_method n, m       # n jest aliasem m.
  }
end
backup(String, :reverse)
"test".original_reverse #=> "test".
```

Jak pamiętasz z podrozdziału 6.1.5, aby usunąć definicję metody, można użyć instrukcji `undef`. Sposób ten działa tylko wtedy, gdy masz możliwość wstawienia nazwy metody do usunięcia bezpośrednio do kodu w postaci identyfikatora. Aby dynamicznie usunąć metodę, której nazwa została obliczona przez program, do wyboru są dwie metody — `remove_method` i `undef_method`. Obie są prywatnymi metodami klasy `Module`. Pierwsza z nich usuwa definicję metody z bieżącej klasy. Jeśli w nadklasie istnieje inna wersja tej metody, zostanie ona odziedziczona. Metoda `undef_method` jest surowsza — blokuje wszelkie wywołania wyznaczonej metody poprzez obiekt tej klasy, nawet jeśli istnieje odziedziczona wersja tej metody.

Aby po zdefiniowaniu klasy zapobiec wszelkim jej modyfikacjom, należy wywołać metodę `freeze` tej klasy. Po zamrożeniu klasa nie może być modyfikowana.

8.4.5. Obsługa niezdefiniowanych metod

Kiedy algorytm rozstrzygania nazw (zobacz podrozdział 7.8) nie znajdzie jakieś metody, szuka w zamian metody `method_missing`. W jej wywołaniu pierwszy argument jest symbolem odpowiadającym nazwie metody, która nie została znaleziona. Po tym symbolu znajdują się wszystkie argumenty, które miały zostać do tej metody przekazane. Jeśli z wywołaniem tej metody jest związany blok, jest on również przekazywany do metody `method_missing`.

Domyślna implementacja metody `method_missing` w module `Kernel` zgłasza tylko wyjątek `NoMethodError`. Jeśli nie zostanie on przechwycony, spowoduje zamknięcie programu i wyświetlenie komunikatu o błędzie, czyli nastąpi dokładnie to, czego należy się spodziewać w przypadku braku metody.

Dzięki zdefiniowaniu własnej metody `method_missing` w swojej klasie można obsłużyć każdy rodzaj wywołania metod na rzecz obiektów tej klasy. Metoda `method_missing` jest jedną z najpotężniejszych funkcji programowania dynamicznego w języku Ruby i jedną z najczęściej używanych technik metaprogramowania. Przykłady jej użycia przedstawione zostaną w dalszej części tego rozdziału. Poniższy przykładowy kod dodaje metodę `method_missing` do klasy `Hash`. Pozwala na sprawdzenie lub ustawienie wartości dowolnego wyznaczonego klucza, tak jakby ten klucz był nazwą metody:

```
class Hash
  # Pozwala na sprawdzanie i ustawianie wartości kluczy, jakby były atrybutami.
  # Symulujemy gettery i settery atrybutów dla każdego klucza.
  def method_missing(key, *args)
    text = key.to_s
    if text[-1, 1] == "="
      self[text.chop.to_sym] = args[0] # Jeżeli klucz kończy się znakiem =, ustawia wartość.
    else
      self[key] # Usunięcie znaku = z klucza.
    end
  end
  h = {} # Utworzenie pustego obiektu.
  h.one = 1 # To samo, co h[:one] = 1.
  puts h.one # Drukuje 1. To samo, co puts h[:one].
```

8.4.6. Ustawianie widoczności metod

W podrozdziale 7.2 wprowadzone zostały metody `public`, `private` i `protected`. Wyglądają one jak słowa kluczowe, ale w rzeczywistości są prywatnymi metodami egzemplarza zdefiniowanymi w klasie `Module`. Metody te są zazwyczaj używane jako statyczna część definicji klasy. Jednak w połączeniu z metodą `class_eval` można używać ich także dynamicznie:

```
String.class_eval { private :reverse }
"witaj".reverse #NoMethodError: private method 'reverse'.
```

Metody `private_class_method` i `public_class_method` są podobne, z tym, że działają na metodach klasowych i same są publiczne:

```
# Czyni wszystkie metody klasy Math prywatnymi.
# Teraz, aby wywołać metodę modułu Math, musimy go domieszać.
Math.private_class_method *Math.singleton_methods
```

8.5. Metody zwrotne

Klasy `Module`, `Class` i `Object` implementują kilka **metod zwrotnych**. Metody te nie są standardowo zdefiniowane, ale jeśli programista zdefiniuje je dla modułu, klasy lub obiektu, będą wywoływanie w odpowiedzi na określone zdarzenia. Pozwala to na rozszerzenie zachowania języka Ruby przy tworzeniu podklas, dołączaniu modułów czy definiowaniu metod. Nazwy metod zwrotnych (poza kilkoma wycofywanymi, które nie zostały tu opisane) kończą się literami „`ed`”.

Kiedy definiowana jest nowa klasa, Ruby wywołuje metodę klasową `inherited` na rzecz nadklasy tej nowej klasy, przekazując jej obiekt jako argument. Dzięki temu klasy mogą dodać zachowania do swoich potomków lub nakładać na nich ograniczenia. Przypomnijmy, że metody klasowe są dziedziczone, a więc metoda `inherited` jest wywoływana, jeśli jest zdefiniowana przez którykolwiek przodka nowo tworzonej klasy. Aby dostawać powiadomienia o wszystkich nowych klasach, należy zdefiniować metodę `Object.inherited`:

```
def Object.inherited(c)
  puts "class #{c} < #{self}"
end
```

Kiedy do klasy lub modułu dołączany jest moduł, następuje wywołanie metody klasowej `included` dołączanego modułu z obiektem klasy lub modułu, do którego został on dołączony, jako argumentem. Dzięki temu można w dołączonym module wzbogacić lub dowolnie zmienić tę klasę — w efekcie umożliwia to modułowi zdefiniowanie metody `include` o dowolnym działaniu. Poza dodawaniem metod do klasy, do której jest dołączany, moduł z metodą `included` może także modyfikować istniejące metody tej klasy, na przykład:

```
module Final          # Nie można tworzyć podklas klasy dołączającej moduł Final.
  def self.included(c) # W przypadku dołączenia tego modułu do klasy c,
    c.instance_eval do # zdefiniuj metodę klasową klasy c
      def inherited(sub) # wykrywającą podklasy i
        raise Exception, # anulującą operacje ich tworzenia za pomocą wyjątku.
          "Próba utworzenia podklasy #{sub} klasy finalnej #{self}"
      end
    end
  end
end
```

Podobnie jeśli moduł zawiera definicję metody klasowej o nazwie `extended`, metoda ta będzie wywoływana zawsze wtedy, gdy moduł ten będzie używany do rozszerzania obiektu (za pomocą metody `Object.extend`). Oczywiście argumentem metody `extended` będzie obiekt, który był rozszerzany. Metoda ta może wykonywać dowolne działania na rzecz tego obiektu.

Poza metodami zwrotnymi śledzącymi klasy i dołączane do nich moduły istnieją też metody zwrotne śledzące metody klas i modułów oraz metody singletonowe dowolnych obiektów. Metodę `method_added` można zdefiniować w każdej klasie i każdym module. Będzie ona

wtedy wywoływana za każdym razem, gdy będzie definiowana metoda egzemplarza tej klasy lub tego modułu:

```
def String.method_added(name)
  puts "Nowa metoda egzemplarza #{name} została dodana do klasy String."
end
```

Należy zauważyć, że metoda klasowa `method_added` jest dziedziczona przez podklasy klasy, w której została zdefiniowana. Nie jest do niej jednak przekazywany żaden argument, przez co nie ma sposobu na stwierdzenie, czy metoda o danej nazwie została dodana do klasy definiującej metodę `method_added`, czy do podklasy tej klasy. Problem ten można obejść, definiując metodę `inherited` w klasie, która zawiera definicję metody `method_added`. Metoda `inherited` może wtedy zdefiniować `method_added` w każdej z podklaśc.

Jeśli dla któregoś obiektu zostanie zdefiniowana metoda singletonowa, na rzecz tego obiektu jest wywoływana metoda `singleton_method_added` przekazująca nazwę nowej metody. Pamiętaj, że dla klas metody singletonowe są metodami klasowymi:

```
def String.singleton_method_added(name)
  puts "Nowa metoda klasowa #{name} została dodana do klasy String."
end
```

Co ciekawe, Ruby wywołuje metodę `singleton_method_added` także w wyniku zdefiniowania po raz pierwszy samej tej metody zwrotnej. Poniżej przedstawiony jest jeszcze jeden sposób użycia tej metody. Tym razem metoda `singleton_method_added` jest zdefiniowana jako metoda egzemplarza dowolnej klasy dołączającej jakiś moduł. Jest powiadamiana o wszystkich metodach singletonowych dodawanych do obiektów tej klasy:

```
# Doliczenie tego modulu do klasy powoduje, ze do jej egzemplarzy nie možna
# dodawać metod singletonowych. Wszystkie dodane metody singletonowe są
# usuwane.
module Strict
  def singleton_method_added(name)
    STDERR.puts "Ostrzeżenie: metoda singletonowa #{name} została dodana do obiektu
Strict."
    eigenklass = class << self; self; end
    eigenklass.class_eval { remove_method name }
  end
end
```

Poza metodami `method_added` i `singleton_method_added` istnieją też metody zwrotne pozwalające śledzić, kiedy metody egzemplarza lub singletonowe są usuwane lub oddefiniowywane. Kiedy w module lub klasie jest usuwana lub oddefiniowywana jakaś metoda, następuje wywołanie metod klasowych `method_removed` lub `method_undefined` na rzecz tego modułu. Usunięcie bądź oddefiniowanie metody singletonowej obiektu powoduje wywołanie metody `singleton_method_removed` lub `singleton_method_undefined` na rzecz tego obiektu.

Na koniec warto zwrócić uwagę, że opisane gdzie indziej w tym rozdziale metody `method_missing` i `const_missing` także działaniem przypominają metody zwrotne.

8.6. Śledzenie

Język Ruby udostępnia kilka narzędzi pozwalających śledzić wykonywanie programów. Ich zastosowanie skupia się głównie na usuwaniu błędów z kodu i drukowaniu przydatnych komunikatów o błędach. Dwa najprostsze z tych narzędzi to słowa kluczowe `_FILE_` i `_LINE_`. Wartością wyrażeń tych słów kluczowych jest zawsze nazwa pliku i numer wiersza, w którym

sie znajdują. Dzięki nim można wygenerować komunikat o błędzie podający, gdzie dokładnie ten błąd wystąpił:

```
STDERR.puts "#{__FILE__}:#{__LINE__}: nieprawidłowe dane"
```

Na marginesie warto zauważyc, że metody Kernel.eval, Object.instance_eval i Module.`__class_eval` przyjmują nazwy plików (lub inne łańcuchy) i numery wiersza jako dwa ostatnie argumenty. Wykonując kod pobrany z jakiegoś pliku, można za pomocą tych argumentów określić wartości słów kluczowych `__FILE__` i `__LINE__` do wykonania.

Z pewnością każdy zauważył, że kiedy zgłoszony wyjątek nie zostanie obsłużony, komunikat o błędzie wydrukowany w konsoli zawiera informacje o nazwie pliku i numerze wiersza. Informacje te są generowane przy użyciu słów kluczowych `__FILE__` i `__LINE__`. Każdy obiekt klasy Exception zawiera informacje o stosie wywołań, które pokazują, gdzie został on zgłoszony, gdzie została wywołana metoda, która zgłosiła wyjątek itd. Metoda Exception.`__backtrace` zwraca tablicę łańcuchów zawierającą te informacje. Pierwszy element tej tablicy określa lokalizację, gdzie wystąpił wyjątek, a każdy kolejny element to jeden poziom stosu wyżej.

Aby uzyskać stos wywołań, nie trzeba jednak zgłaszać wyjątku. Metoda Kernel.caller zwraca aktualny stan stosu wywołań w takim samym formacie jak metoda Exception.backtrace. Wywołana bez żadnych argumentów metoda caller zwraca stos wywołań; jego pierwszym elementem jest metoda, która wywołała metodę wywołującą metodę caller. To znaczy element `caller[0]` wskazuje lokalizację, w której została wywołana bieżąca metoda. Metodę caller można także wywołać z argumentem określającym, ile poziomów stosu wywołań usunąć, licząc od początku zwróconych informacji o tym stosie. Domyślna wartość to 1, a element `caller(0)[0]` określa lokalizację, w której została wywołana metoda caller. Oznacza to zatem, że `caller[0]` jest tym samym co `caller(0)[1]`, a `caller(2)` tym samym co `caller[1..-1]`.

Stosy wywołań zwracane przez metody Exception.backtrace i Kernel.caller zawierają także nazwy metod. Przed pojawiением się Ruby 1.9, aby wydobyć nazwy metod, trzeba było przeanalizować łańcuchy stosu wywołań. Natomiast w Ruby 1.9 nazwę aktualnie wykonywanej metody (w postaci symbolu) można sprawdzić za pomocą metody Kernel.`__method__` lub jej synonimu Kernel.`__callee__`. Metoda `__method__` jest przydatna w połączeniu ze słowami kluczowymi `__FILE__` i `__LINE__`:

```
raise "Błąd w #__method__ at #__FILE__:#__LINE__"
```

Pamiętaj, że metoda `__method__` zwraca oryginalną nazwę metody, nawet jeśli została ona wywołana przez alias.

Zamiast tylko wyświetlać nazwę pliku i numer wiersza, w którym wystąpił błąd, można pójść o krok dalej i wyświetlić ten wiersz kodu. Jeśli w programie jest zdefiniowana stała globalna o nazwie `SCRIPT_LINES__` połączona znakiem równości z tablicą asocjacyjną, metody require i load dodają do tej tablicy asocjacyjnej wpis odpowiadający każdemu załadowanemu przez nie plikowi. Kluczami w tej tablicy są nazwy plików, a wartościami tablice zawierające wiersze z tych plików. Aby dodać do tej tablicy asocjacyjnej plik główny (zamiast plików do niego dołączanych), należy napisać:

```
SCRIPT_LINES__ = {__FILE__ => File.readlines(__FILE__)}
```

Dzięki temu w każdym miejscu programu można uzyskać aktualny wiersz kodu za pomocą poniższego wyrażenia:

```
SCRIPT_LINES__[__FILE__][__LINE__-1]
```

Ruby pozwala na śledzenie przypisań do zmiennych globalnych za pomocą metody `Kernel.<trace_var>`. Należy do niej przekazać symbol nazwy zmiennej globalnej i łańcuch lub blok kodu. Kiedy wartość zmiennej zmieni się, zostanie wykonany podany łańcuch lub wywołany podany blok kodu. Gdy zostanie podany blok kodu, nowa wartość zmiennej jest do niego przekazywana jako argument. Aby zatrzymać śledzenie zmiennej, należy wywołać metodę `Kernel.untrace_var`. W poniższym przykładowym programie warto zwrócić uwagę na użycie elementu `caller[1]` do określenia, w którym miejscu programu został wywołany blok śledzenia zmiennej:

```
# Drukuje komunikat po każdej zmianie wartości zmiennej $SAFE.
trace_var(:$SAFE) { |v|
  puts "Wartość zmiennej $SAFE została ustawiona na #{v} w #{caller[1]}"
}
```

Ostatnia metoda śledząca ma nazwę `Kernel.set_trace_func`. Rejestruje ona obiekt klasy `Proc`, który ma być wywoływany po każdym wierszu kodu. Metoda `set_trace_func` przydaje się w pisaniu modułów debugera pozwalających na wykonywanie programu wiersz po wierszu. Nie będziemy jej jednak szczegółowo opisywać.

8.7. Moduły ObjectSpace i GC

Moduł `ObjectSpace` zawiera kilka przydatnych metod działających na niskim poziomie, które mogą czasami znaleźć zastosowanie w usuwaniu błędów lub metaprogramowaniu. Najbardziej godną uwagi z tych metod jest `each_object` — iterator, który może zwrócić każdy obiekt (lub każdy egzemplarz określonej klasy), o którym wie interpreter:

```
# Drukuje listę wszystkich znanych klas.
ObjectSpace.each_object(Class) { |c| puts c }
```

Metoda `ObjectSpace._id2ref` jest odwrotnością metody `Object.object_id` — pobiera jako argument identyfikator obiektu i zwraca odpowiadający mu obiekt lub zgłasza wyjątek `RangeError`, jeśli nie ma obiektu o takim identyfikatorze.

Metoda `ObjectSpace.define_finalizer` pozwala zarejestrować obiekt klasy `Proc` lub blok kodu, który zostanie wywołany w odpowiedzi na poddanie określonego obiektu działaniu systemu przywracania pamięci. Należy jednak przy rejestracji takiej metody finalizującej zachować ostrożność, ponieważ jej blok nie może używać tego czyszczonego obiektu. Wszystkie wartości wymagane do finalizacji obiektu muszą znajdować się w zakresie bloku finalizującego, aby były dostępne bez wyłuskiwania tego obiektu. Aby usunąć wszystkie bloki finalizujące zarejestrowane dla obiektu, należy użyć metody `ObjectSpace.undefined_finalizer`.

Ostatnia metoda modułu `ObjectSpace` to `ObjectSpace.garbage_collect`. Wymusza ona działanie systemu przywracania pamięci. Funkcjonalność tego systemu jest także dostępna poprzez moduł `GC`. Metoda `GC.start` jest synonimem metody `ObjectSpace.garbage_collect`. System przywracania pamięci można tymczasowo wyłączyć za pomocą metody `GC.disable`, a włączyć za pomocą metody `GC.enable`.

Kombinacja metod `_id2ref` i `define_finalizer` pozwala na zdefiniowanie obiektów „słabych referencji” przechowujących referencje do wartości, nie blokując ich usuwania przez system przywracania pamięci, jeśli nie będzie do nich dostępu w inny sposób. Przykład ich użycia można znaleźć w klasie `WeakRef` w bibliotece standardowej w pliku `lib/weakref.rb`.

8.8. Niestandardowe struktury sterujące

Bloki w języku Ruby w połączeniu ze składnią pozwalającą na opcjonalne stosowanie nawiasów sprawiają, że bardzo łatwo można definiować metody iteracyjne wygładzające i działające jak struktury sterujące. Prostym tego przykładem jest metoda `loop` z modułu `Kernel`. W tym podrozdziale prezentujemy jeszcze trzy tego typu przykłady. W przykładach tych wykorzystywane jest API wątków, a więc do pełnego ich zrozumienia może być konieczne zapoznanie się z podrozdziałem 9.9.

8.8.1. Opóźnianie i powtarzanie wykonania — metody `after` i `every`

W programie na listingu 8.1 zdefiniowano globalne metody `after` i `every`. Każda z nich przyjmuje argument liczbowy reprezentujący liczbę sekund i z każdą z nich powinien być związany blok kodu. Metoda `after` tworzy nowy wątek i natychmiast zwraca reprezentującą go obiekt klasy `Thread`. Nowo utworzony wątek śpi przez wyznaczoną liczbę sekund, a następnie wywołuje (bez żadnych argumentów) podany blok kodu. Metoda `every` działa podobnie, ale wykonuje blok wielokrotnie, oczekując pomiędzy kolejnymi wywołaniami określona liczbę sekund. Drugi argument tej metody jest przekazywany do pierwszego wywołania bloku. Wartość zwrotna każdego z tych wywołań jest przekazywana do kolejnego. Aby zapobiec kolejnym wywołaniom bloku metody `every`, można użyć instrukcji `break`.

Poniżej znajduje się przykładowy fragment programu z wykorzystaniem metod `after` i `every`:

```
require 'afterevery'  
1.upto(5) { |i| after i { puts i } } # Powoli drukuje cyfry od 1 do 5.  
sleep(5) # Czeka 5 sekund.  
every 1, 6..do |count| # Powoli drukuje liczby od 6 do 10.  
  puts count  
  break if count == 10  
  count + 1 # Kolejna wartość zmiennej count.  
end  
sleep(6) # Daje czas wątkowi na zakończenie działania.
```

Wywołanie metody `sleep` na samym końcu kodu zapobiega zamknięciu programu, zanim wątek utworzony przez metodę `every` zakończy odliczanie. Wiedząc już jak działają metody `after` i `every`, można przejść do analizy ich implementacji. Jeśli nie rozumiesz działania metody `Thread.new`, przeczytaj podrozdział 9.9.

Listing 8.1. Metody `after` i `every`

```
#  
# Definicja metod after i every z modułu Kernel, aby opóźniały wykonywanie bloków kodu.  
# Przykłady:  
#  
# after 1 { puts "koniec" }  
# every 60 { redraw_clock }  
#  
# Obie te metody zwracają obiekty klasy Thread. Aby anulować wykonywanie tego kodu,  
# należy wywołać metodę kill na rzecz tych obiektów.  
#  
# Pamiętaj, że jest to bardzo prosta implementacja. Bardziej niezawodna wersja,  
# dla wszystkich zadań używałaby jednego globalnego wątku mierzącego czas oraz  
# umożliwiałaby sprawdzanie wartości opóźnionego bloku i poczekanie  
# na zakończenie wszystkich zadań.  
#  
# Wykonuje blok kodu po oczekaniu wyznaczonej liczby sekund.
```

```

def after(seconds, &block)
  Thread.new do          # Nowy wątek.
    sleep(seconds)        # Pierwsze zaśnięcie.
    block.call            # Wywołanie bloku.
  end                    # Zwrócenie natychmiast obiektu klasy Thread.
end
# Wielokrotnie usypia, a następnie wykonuje blok kodu.
# Przekazuje do bloku wartość przy pierwszym wywołaniu.
# Do kolejnych wywołań przekazuje wartość poprzedniego wywołania.
def every(seconds, value=nil, &block)
  Thread.new do          # Nowy wątek.
    loop do               # Powtarzanie w nieskończoność (lub do instrukcji break w bloku).
      sleep(seconds)      # Zaśnięcie.
      value = block.call(value) # Wywołanie bloku.
    end                  # Następne powtórzenie.
  end                    # Metoda every zwraca obiekt klasy Thread.
end

```

8.8.2. Bezpieczeństwo wątków i synchronizacja bloków

Pisząc program wielowątkowy, należy uważać, aby dwa wątki nie modyfikowały tego samego obiektu w tym samym czasie. Jednym ze sposobów na to jest umieszczenie kodu, który musi być bezpieczny wątkowo w bloku związanym z wywołaniem metody `synchronize` obiektu klasy `Mutex`. Szerzej na ten temat piszemy w podrozdziale 9.9. Na listingu 8.2 posuwamy się o krok dalej i emulujemy słowo kluczowe Javy `synchronized` przy użyciu globalnej metody o nazwie `synchronized`. Metoda ta wymaga jednego argumentu i bloku. Problem w tym, że z obiektami w języku Ruby, w przeciwieństwie do obiektów Javy, nie są związane mutexy. Dlatego zdefiniowana została także metoda egzemplarza o nazwie `mutex` w klasie `Object`. Co ciekawe, w implementacji tej metody użyto `synchronized` w nowej formie słowa kluczowego!

Listing 8.2. Proste zsynchronizowane bloki

```

# Pozyskanie mutexu związanego z obiektem o i wykonanie
# bloku pod ochroną tego mutexu.
# Działa podobnie do słowa kluczowego Javy synchronized.
def synchronized(o)
  o.mutex.synchronize { yield }
end
# Metoda Object.mutex nie istnieje, a więc trzeba ją zdefiniować.
# Metoda ta zwraca unikatowy mutex dla każdego obiektu i dla poszczególnych
# obiektów zwraca zawsze ten sam mutex.
# Mutexy są tworzone leniwie, co wymaga synchronizacji ze względu na
# bezpieczeństwo wątków.
class Object
  # Metoda ta zwraca mutex dla tego obiektu, tworząc go w razie potrzeby.
  # Sztuka polega na tym, aby dwa różne wątki nie wywołyły
  # jej jednocześnie, tworząc dwa różne mutexy.
  def mutex
    # Jeśli ten obiekt posiada już mutex, zostanie on zwrócony.
    return @_mutex if @_mutex

    # W przeciwnym przypadku konieczne jest utworzenie mutexu dla tego obiektu.
    # Aby było bezpiecznie, konieczna jest synchronizacja naszego obiektu klasy.
    synchronize(self.class) {
      # Ponowne sprawdzenie: od czasu wejścia do tego synchronizowanego bloku
      # jakiś inny wątek mógł już utworzyć ten mutex.
      @_mutex = @_mutex || Mutex.new
    }
  end
end

```

```

# Wartością zwrotną jest @_mutex.
end
end
# Zdefiniowana powyżej metoda Object.mutex musi zablokować klasę, jeśli
# obiekt nie ma jeszcze muteksu. Jeżeli klasa nie posiada jeszcze własnego muteksu,
# obiekt tej klasy zostanie zablokowany. Aby zapobiec nieskończonej rekurencji,
# musisz upewnić się, że obiekt ten ma mutex.
Class.instance_eval { @_mutex = Mutex.new }

```

8.9. Brakujące metody i stałe

Metoda `method_missing` jest kluczową częścią algorytmu wyszukiwania metod w języku Ruby (zobacz podrozdział 7.8). Pozwala ona przechwytywać i obsługiwać dowolne wywołania metod na rzecz obiektów. Metoda `constant_missing` z klasy `Module` pełni podobną rolę w algorytmie wyszukiwania stałych. Pozwala obliczyć lub leniwie zainicjować stałe w locie. Prezentowane dalej przykłady demonstrują użycie obu tych metod.

8.9.1. Stałe punktów kodowych Unicode i metoda `const_missing`

W programie na listingu 8.3 zdefiniowano moduł o nazwie `Unicode`, który definiuje stałe (łańcuch w kodowaniu UTF-8) dla wszystkich punktów kodowych Unicode, od U+0000 do U+10FFFF. Jedyny praktyczny sposób na poradzenie sobie z tak dużą liczbą stałych polega na użyciu metody `const_missing`. W kodzie tym zrobiono założenie, że jeśli istnieje jedno odwołanie do stałej, możliwe jest, że będzie ich więcej. W związku z tym metoda `const_missing` wywołuje metodę `Module.const_set` definiującą prawdziwą stałą odnoszącą się do każdej z wyliczonych wartości.

Listing 8.3. Stałe punktów kodowych Unicode i metoda `const_missing`

```

# Niniejszy moduł dostarcza stałych definiujących lańcuchy UTF-8 dla
# wszystkich punktów kodowych Unicode. Definiuje je leniwie za pomocą metody const_missing.
# Przykłady:
#   copyright = Unicode::U00A9
#   euro = Unicode::U20AC
#   infinity = Unicode::U221E
module Unicode
  # Niniejsza metoda pozwala na leniwe definiowanie stałych punktów kodowych Unicode.
  def self.const_missing(name) # Niezdefiniowana stała przekazana jako symbol.
    # Sprawdzenie, czy nazwa stałej ma poprawny format —
    # musi być wielka litera U z liczbą szesnastkową z przedziału 0000 - 10FFFF.
    if name.to_s =~ /(^U([0-9a-fA-F]{4,5}|10[0-9a-fA-F]{4})$)/
      # $1 to dopasowana liczba szesnastkowa. Zostaje ona przekonwertowana na liczbę całkowitą.
      codepoint = $1.to_i(16)
      # Konwersja liczby na lańcuch UTF-8 za pomocą metody Array.pack.
      utf8 = [codepoint].pack("U")
      # Sprawienie, że nie można modyfikować tego lańcucha UTF-8.
      utf8.freeze
      # Definicja prawdziwej stałej, aby przyspieszyć kolejne wyszukiwania, i zwrócenie
      # tym razem tekstu UTF-8.
      const_set(name, utf8)
    else
      # Zgłoszenie wyjątku dla stałych w nieprawidłowym formacie.
      raise NameError, "Niezainicjowana stała: Unicode::#{name}"
    end
  end
end

```

8.9.2. Śledzenie wywołań metod za pomocą metody `method_missing`

Wcześniej w tym rozdziale zademonstrowany został sposób rozszerzenia klasy Hash za pomocą metody `method_missing`. W programie z listingu 8.4 zademonstrowane jest natomiast zastosowanie metody `method_missing` do delegacji dowolnie wybranych wywołań na rzecz jakiegoś obiektu do innego obiektu. Tym razem robione jest to, aby uzyskać informacje ze śledzenia obiektu.

Na listingu 8.4 zdefiniowano metodę egzemplarza `Object.trace` i klasę `TracedObject`. Metoda `trace` zwraca egzemplarz klasy `TracedObject`, który za pomocą metody `method_missing` przechwytuje wywołania, śledzi je i oddelegowuje do śledzonego obiektu. Można jej użyć w następujący sposób:

```
a = [1,2,3].trace("a")
a.reverse
puts a[2]
puts a.fetch(3)
```

Powyższy kod zwróci następujące dane ze śledzenia:

```
Wywołanie: a.reverse() at trace1.rb:66
Zwarcanie: [3, 2, 1] from a.reverse to trace1.rb:66
Wywołanie: a.fetch(3) at trace1.rb:67
Zgłoszenie: IndexError:index 3 out of array z a.fetch
```

Zauważ, że poza metodą `method_missing` program na listingu 8.4 demonstruje także metody `Module.instance_methods`, `Module.undef_method` i `Kernel.caller`.

Listing 8.4. Śledzenie wywołań metod za pomocą metody `method_missing`

```
# Metoda trace dowolnego obiektu tworzy nowy obiekt, który zachowuje się
# dokładnie tak samo jak oryginal, ale śledzi wszystkie wywołania metod
# na rzecz tego obiektu. W przypadku śledzenia więcej niż jednego obiektu,
# należy podać nazwę, która ma zostać użyta w danych wyjściowych. Domyślnie
# komunikaty będą wysypane do strumienia STDERR, ale można wskazać dowolny strumień
# (lub dowolny obiekt przyjmującyłańcuchy jako argumenty do <<).
class Object
  def trace(name="", stream=STDERR)
    # Zwraca śledzący i delegujący obiekt klasy TracedObject.
    TracedObject.new(self, name, stream)
  end
end
# Niniejsza klasa śledzi wywołania metod za pomocą metody method_missing i
# deleguje je do jakiegoś innego obiektu. Usuwa większość własnych
# metod egzemplarza, aby nie wchodziły w konflikty z metodą method_missing.
# Zauważ, że śledzone będą tylko metody wywoływane poprzez obiekt klasy TracedObject.
# Jeśli obiekt delegowany wywołuje metodę na swoją rzecz, śledzenia te nie
# są śledzone.
class TracedObject
  # Wszystkie małe ważne publiczne metody egzemplarza zostają oddefiniowane.
  # Zauważ użycie metod Module.instance_methods i Module.undef_method.
  instance_methods.each do |m|
    m = m.to_sym # Ruby 1.8 zwracałałańcuchy zamiast symboli.
    next if m == :object_id || m == :__id__ || m == :__send__
    undef_method m
  end
  # Inicjalizacja egzemplarza klasy TracedObject.
  def initialize(o, name, stream)
    @o = o           # Obiekt, do którego delegujemy.
    # Ostatni argument jest opcjonalny i służy do zapisania nazwy obiektu.
  end
end
```

```

@n = name          # Nazwa obiektu, która ma pojawiać się w komunikatach.
@trace = stream    # Miejsce, do którego mają być wysyłane komunikaty.
end
# To jest kluczowa metoda klasy TracedObject. Jest wywoływana dla
# każdego wywołania metody na rzecz obiektu klasy TracedObject.
def method_missing(*args, &block)
  m = args.shift      # Pierwszy argument jest nazwą metody.
  begin
    # Śledzenie wywołań tej metody.
    arglist = args.map {|a| a.inspect}.join(' ', ' ')
    @trace << "Wywoływanie: #{@n}.#{m}(#{arglist}) w #{caller[0]}\n"
    # Wywołanie metody na rzecz delegowanego obiektu i odebranie wartości zwrotnej.
    r = @o.send m, *args, &block
    # Śledzenie normalnego zwrotu metody.
    @trace << "Zwracanie: #{r.inspect} z #{@n}.#{m} do #{caller[0]}\n"
    # Zwrócenie wartości zwrotnej przez obiekt delegowany.
    r
  rescue Exception => e
    # Śledzenie nienormalnego zwrotu metody.
    @trace << "Zgłoszenie: #{e.class}:#{e} z #{@n}.#{m}\n"
    # Ponowne zgłoszenie wyjątku zgłoszonego przez obiekt delegowany.
    raise
  end
  # Zwrócenie obiektu, do którego delegujesz.
  def __delegate
    @o
  end
end

```

8.9.3. Obiekty synchronizowane przez delegację

Na listingu 8.2 przedstawiona została globalna metoda o nazwie `synchronized` przyjmująca obiekt i wykonującą blok kodu pod ochroną muteksu związanego z tym obiektem. Większość kodu tego listingu zajmuje implementacja metody `Object.mutex`. Metoda `synchronized` była bardzo prosta:

```

def synchronized(o)
  o.mutex.synchronize { yield }
end

```

Na listingu 8.5 ta metoda została zmodyfikowana, aby w przypadku wywołania jej bez bloku zwracała obiekt klasy `SynchronizedObject` opakowujący ten obiekt. `SynchronizedObject` to delegacyjna klasa osłonowa oparta na metodzie `method_missing`. Jest w dużym stopniu podobna do klasy `TracedObject` z listingu 8.4, ale została napisana jako podklasa klasy `BasicObject` Ruby 1.9. Dzięki temu nie ma potrzeby jawnego usuwania metody egzemplarza klasy `Object`. Zauważ, że kod z tego listingu nie działa samodzielnie — wymaga uprzedniego zdefiniowania metody `Object.mutex`.

Listing 8.5. Synchronizacja metod za pomocą metody `method_missing`

```

def synchronized(o)
  if block_given?
    o.mutex.synchronize { yield }
  else
    SynchronizedObject.new(o)
  end
end
# Delegacyjna klasa osłonowa zapewniająca bezpieczeństwo wątkom za pomocą metody method_missing.
# Zamiast rozszerzyć klasę Object i delegować metody, rozszerzasz klasę

```

```

# BasicObject, która jest zdefiniowana w Ruby 1.9. Klasa BasicObject nie
# dziedziczy po klasie Object ani module Kernel, a więc jej obiekty nie mogą
# wywoływać żadnych metod najwyższego poziomu — po prostu ich tam nie ma.
class SynchronizedObject < BasicObject
  def initialize(o); @delegate = o; end
  def __delegate; @delegate; end
  def method_missing(*args, &block)
    @delegate.mutex.synchronize {
      @delegate.send *args, &block
    }
  end
end

```

8.10. Dynamiczne tworzenie metod

Jedną z najważniejszych technik metaprogramowania jest używanie metod do tworzenia innych metod. Przykładami na to są metody `attr_reader` i `attr_accessor` (zobacz podrozdział 7.1.5). Te prywatne metody egzemplarza klasy `Module` są używane w definicjach klas jak słowa kluczowe. Jako argumenty przyjmują nazwy atrybutów i przy użyciu tych nazw dynamicznie tworzą nowe metody. Przedstawione dalej przykładowe programy są wariantami tych metod tworzących metody dostępu do atrybutów i demonstrują dwa różne sposoby na dynamiczne tworzenie metod tego typu.

8.10.1. Definiowanie metod za pomocą metody `class_eval`

Na listingu 8.6 zdefiniowano dwie prywatne metody egzemplarza klasy `Module` o nazwie `readonly` i `readwrite`. Działają one tak samo jak metody `attr_reader` i `attr_accessor`, a zostały tu przedstawione tylko po to, aby zademonstrować, jak się je implementuje. Jest to bardzo proste — metody te najpierw budują łańcuch kodu Ruby zawierający instrukcje `def` potrzebne do zdefiniowania odpowiednich metod dostępowych. Następnie wykonują ten łańcuch za pomocą metody `class_eval` (została opisana wcześniej w tym rozdziale). Z takim sposobem użycia metody `class_eval` wiąże się nieznaczny narzut spowodowany przetwarzaniem łańcucha kodu. Korzyść jest taka, że zdefiniowane w ten sposób metody same nie muszą korzystać z żadnego API refleksji — mogą sprawdzać lub ustawać wartości zmiennych bezpośrednio.

Listing 8.6. Metody atrybutowe definiowane za pomocą metody `class_eval`

```

class Module
  private          # Poniższe metody są prywatne.
  # Ta metoda działa jak metoda attr_reader, ale ma krótszą nazwę.
  def readonly(*syms)
    return if syms.size == 0 # Jeśli nie ma argumentów, nie robi nic.
    code = ""               # Na początku jest pusty łańcuch.
    # Generuje łańcuch kodu Ruby definiujący metody odczytujące atrybuty.
    # Zauważ interpolację symbolu do łańcucha kodu.
    syms.each do |s|         # Dla każdego symbolu
      code << "def #{s}; @#{s}; end\n" # definiowana jest metoda.
    end
    # Tworzenie metod egzemplarza z wygenerowanego kodu za pomocą metody class_eval.
    class_eval code
  end
  # Ta metoda działa jak metoda attr_accessor, ale ma krótszą nazwę.
  def readwrite(*syms)

```

```

return if syms.size == 0
code = ""
syms.each do |s|
  code << "def #{s}; #{@{s}} end\n"
  code << "def #{s}=(value); #{@{s}} = value; end\n"
end
class_eval code
end
end

```

8.10.2. Definiowanie metod za pomocą metody `define_method`

Listing 8.7 przedstawia odmienne podejście do akcesorów atrybutów. Metoda `attribute` jest podobna do metody `readwrite` z listingu 8.6. Zamiast przyjmować dowolną liczbę nazw atrybutów jako argumenty, przyjmuje jeden obiekt tablicy asocjacyjnej. Kluczami w tej tablicy powinny być nazwy atrybutów, a wartościami domyślne wartości tych atrybutów. Metoda `clas_attrs` działa tak samo jak metoda `attributes`, ale definiuje atrybuty klas zamiast atrybutów egzemplarzy.

Przypomnijmy, że w języku Ruby można opuścić klamry literalów haszowych, jeśli są one ostatnim argumentem w wywołaniu metody. W związku z tym metodę `attributes` można wywołać następująco:

```

class Point
  attributes :x => 0, :y => 0
end

```

W Ruby 1.9 można zastosować bardziej zwięzłą składnię:

```

class Point
  attributes x:0, y:0
end

```

Jest to jeszcze jeden przykład potwierdzający, że elastyczna składnia języka Ruby pozwala tworzyć metody, które zachowują się jak słowa kluczowe.

Implementacja metody `attributes` na listingu 8.7 różni się od implementacji metody `readwrite` na listingu 8.6. Zamiast definiować łańcuch kodu i wykonywać go za pomocą metody `class_eval`, metoda `attributes` definiuje ciało akcesorów atrybutów w bloku i definiuje te metody za pomocą metody `define_method`. Ponieważ niniejsza technika definiowania metod nie pozwala na interpolację identyfikatorów bezpośrednio do ciała metody, konieczne jest posłużenie się metodami refleksyjnymi, takimi jak `instance_variable_get`. Z tego względu akcesory definiowane przy użyciu metody `attributes` mogą być wolniejsze od akcesorów zdefiniowanych przy użyciu metody `readwrite`.

Interesujące jest to, że metoda `attributes` nie przechowuje jawnie domyślnych wartości atrybutów w żadnej zmiennej. W zamian wartość każdego atrybutu jest przechwytywana przez zakres bloku użytego do zdefiniowania metody (więcej informacji na temat tego typu domknięć znajduje się w podrozdziale 6.6).

Metoda `classAttrs` definiuje atrybuty klas w bardzo prosty sposób — wywołuje metodę `attributes` na rzecz klasy `eigenclass` klasy. Oznacza to, że powstałe w ten sposób metody używają zmiennych egzemplarza klasy (zobacz podrozdział 7.1.16) zamiast zwykłych zmiennych klasowych.

Listing 8.7. Definiowanie metod atrybutowych za pomocą metody `define_method`

```
class Module
  # Niniejsza metoda definiuje metody odczytu i zapisu atrybutów dla wyznaczonych
  # atrybutów, ale przyjmuje jako argument tablicę asocjacyjną rzuającą nazwy atrybutów
  # na ich wartości domyślne. Wygenerowane w ten sposób metody odczytu atrybutów zwracają
  # swoje wartości domyślne, jeśli zmieniona egzemplarza nie jest jeszcze zdefiniowana.
  def attributes(hash)
    hash.each_pair do |symbol, default|
      # Dla każdej pary atrybut/wartość domyślna.
      getter = symbol
      # Nazwa metody sprawdzającej.
      setter = :"#{symbol}="
      # Nazwa metody ustawiającej.
      variable = "@#{symbol}"
      # Nazwa zmiennej egzemplarza.
      define_method getter do
        # Definicja metody sprawdzającej.
        if instance_variable_defined? variable
          instance_variable_get variable # Zwraca zmienną, jeśli jest zdefiniowana.
        else
          default
        end
      end
      define_method setter do |value|
        # Definicja metody ustawiającej.
        instance_variable_set variable,
          value
        # Ustawienie zmiennej egzemplarza
        # na wartość argumentu.
      end
    end
  end
  # Metoda ta działa jak metoda attributes, ale definiuje metody klasowe,
  # wywołując metodę attributes na rzecz klasy eigenclass zamiast self.
  # Zdefiniowane metody używają zmiennej egzemplarza klasy
  # zamiast zwykłych zmiennych klasowych.
  def class attrs(hash)
    eigenclass = class << self; self; end
    eigenclass.class_eval { attributes(hash) }
  end
  # Obie metody są prywatne.
  private :attributes, :class attrs
end
```

8.11. Tworzenie łańcuchów aliasów

Przekonałeś się już, że metaprogramowanie w języku Ruby często wymaga dynamicznego definiowania metod. Niemniej powszechnie jest dynamiczne **modyfikowanie** metod. Technika modyfikacji metod nazywa się **tworzeniem łańcuchów aliasów**¹ i wygląda następująco:

- Utworzenie aliasu metody, która ma zostać zmodyfikowana. Alias ten jest nazwą dla niezmodyfikowanej wersji tej metody.
- Zdefiniowanie nowej wersji metody. Powinna ona wywoływać niezmodyfikowaną wersję poprzez alias, ale przed i po zrobieniu tego może dodać dowolne własne funkcje.

Czynności te można wykonywać wielokrotnie (pod warunkiem że za każdym razem używa się innego aliasu), tworząc łańcuch metod i aliasów.

W niniejszym podrozdziale przedstawione zostaną trzy przykłady łańcuchów aliasów. W pierwszym łańcuch aliasów jest tworzony statycznie, tj. przy użyciu zwykłych instrukcji `alias` i `def`.

¹ Technika ta jest także nazywana po angielsku **monkey chaining**, ale ponieważ termin ten był początkowo używany z nutką drwiny, staramy się go unikać. Czasami jako humorystycznej alternatywy używa się też angielskiego określenia **duck punching**.

Przykłady drugi i trzeci są bardziej dynamiczne — tworzą łańcuchy aliasów arbitralnie wyznaczanych metod przy użyciu metod `alias_method`, `define_method` i `class_eval`.

8.11.1. Śledzenie załadowanych plików i zdefiniowanych klas

Program na listingu 8.8 śledzi wszystkie załadowane pliki i zdefiniowane klasy w programie. Przed zamknięciem drukuje raport. Kod ten można wykorzystać w innym programie, aby lepiej zrozumieć, co się w nim dzieje. Jednym ze sposobów na użycie tego kodu jest umieszczenie na początku programu poniższego wiersza:

```
require 'classtrace'
```

Łatwiej jednak jest użyć opcji `-r` interpretera:

```
ruby -rclasstrace my_program.rb --traceout /tmp/trace
```

Opcja `-r` powoduje załadowanie wyznaczonej biblioteki przed uruchomieniem programu. Więcej informacji o argumentach wiersza poleceń znajduje się w podrozdziale 10.1.

Program na listingu 8.8 wykorzystuje statyczną technikę tworzenia aliasów do śledzenia wszystkich wywołań metod `Kernel.require` i `Kernel.load`. Metoda zwrotna `Object.inherited` śledzi definicje nowych klas. Natomiast metoda `Kernel.at_exit` wykonuje blok kodu, kiedy program kończy działanie (w tym przypadku można by było także użyć instrukcji `END` opisanej w podrozdziale 5.7). Poza utworzeniem łańcuchów aliasów metod `require` i `load` i zmodyfikowaniem metody `Object.inherited` jedyną modyfikacją globalnej przestrzeni nazw dokonaną przez ten kod jest zdefiniowanie modułu o nazwie `ClassTrace`. Cały stan wymagany do śledzenia jest zapisany w stałych w tym module, dzięki czemu nie zaśmieca się przestrzeni nazw zmiennymi globalnymi.

Listing 8.8. Śledzenie załadowanych plików i zdefiniowanych klas

```
# Niniejszy moduł ma za zadanie przechowywać informacje o stanie globalnym, dzięki czemu
# nie ma potrzeby modyfikować globalnej przestrzeni nazw bardziej niż jest to konieczne.
module ClassTrace
  # Niniejsza tablica przechowuje listę załadowanych plików i zdefiniowanych klas.
  # Każdy element jest podtablicą zawierającą zdefiniowaną klasę lub
  # załadowany plik oraz ramkę stosu, na którym klasę tę zdefiniowano lub załadowano plik.
  T = [] # Tablica przechowująca załadowane pliki.
  # Teraz zostanie zdefiniowana stała OUT określająca, gdzie mają zostać wysłane dane ze śledzenia.
  # Domyslnie jest to strumień STDERR, ale można to zmienić za pomocą argumentów wiersza poleceń.
  if x = ARGV.index("--traceout") # Jeśli argument istnieje,
    OUT = File.open(ARGV[x+1], "w") # zostaje otwarty wyznaczony plik,
    ARGV[x, 2] = nil # a argumenty zostają usunięte.
  else
    OUT = STDERR # W przeciwnym przypadku stosowana jest wartość domyślna.
  end
  # Tworzenie łańcucha aliasów — krok 1.: definicja aliasów dla oryginalnych metod.
  alias original_require require
  alias original_load load
  # Tworzenie łańcucha aliasów — krok 2.: definicja nowych wersji tych metod.
  def require(file)
    ClassTrace::T << [file, caller[0]] # Zapamiętuje, co zostało gdzie załadowane.
    original_require(file) # Wywołanie oryginalnej metody.
  end
  def load(*args)
    ClassTrace::T << [args[0], caller[0]] # Zapamiętanie, co zostało gdzie załadowane.
    original_load(*args) # Wywołanie oryginalnej metody.
  end
end
```

```

# Niniejsza metoda zwrotna jest wywoływana w odpowiedzi na definicję każdej klasy.
def Object.inherited(c)
  ClassTrace::T << [c,caller[0]]           # Zapamiętanie, co zostało gdzie zdefiniowane.
end
# Metoda Kernel.at_exit rejestruje blok, który ma być wykonywany przed zamknięciem programu.
# Za jej pomocą program raportuje zebrane informacje o plikach i klasach.
at_exit {
  o = ClassTrace::OUT
  o.puts "="*60
  o.puts "Załadowane pliki i zdefiniowane klasy:"
  o.puts "="*60
  ClassTrace::T.each do |what,where|
    if what.is_a? Class # Raport o zdefiniowanej klasie (z hierarchią).
      o.puts "Zdefiniowano: #{what.ancestors.join('<-')} w #{where}"
    else                # Raport o załadowanym pliku.
      o.puts "Załadowano: #{what} w #{where}"
    end
  end
}

```

8.11.2. Tworzenie łańcuchów metod dla bezpieczeństwa wątków

W dwóch z prezentowanych do tej pory przykładowych programach dbano o bezpieczeństwo wątków. Na listingu 8.2 zdefiniowano metodę `synchronized` (na bazie metody `Object.mutex`), która wykonywała obiekty pod ochroną obiektu klasy `Mutex`. Następnie na listingu 8.5 przeddefiniowano metodę `synchronized`, aby w przypadku wywołania bez bloku zwracała obiekt klasy `SynchronizedObject` opakowujący obiekt, chroniąc dostęp do metod wywoływanego przez ten obiekt opakowujący. Teraz w programie na listingu 8.9 ponownie wzbogaca się metodę `synchronized`, aby, jeśli zostanie wywołana wewnętrz definiacji klasy lub modułu, tworzyła łańcuch aliasów wyznaczonych metod w celu dodania synchronizacji.

Tworzeniem łańcucha aliasów zajmuje się metoda `Module.synchronize_method`, która z kolei za pomocą metody pomocniczej `Module.create_alias` definiuje odpowiedni alias dla każdej podanej metody (wliczając metody operatorowe typu `+`).

Po zdefiniowaniu tych nowych metod klasy `Module` ponownie zostaje przeddefiniowana metoda `synchronized`. Kiedy zostaje ona wywołana wewnętrz klasy lub modułu, wywołuje metodę `synchronize_method` na rzecz każdego z przekazanych do niej symboli. Interesujące jest to, że można ją wywołać także bez żadnych argumentów. Wtedy dodaje synchronizację do tej metody egzemplarza, której definicja występuje jako następna (do odbierania powiadomień o dodawanych nowych metodach wykorzystuje metodę zwrotną `method_added`). Zauważ, że kod tego programu wykorzystuje metodę `Object.mutex` z listingu 8.2 i klasę `SynchronizedObject` z listingu 8.5.

Listing 8.9. Tworzenie łańcucha aliasów dla bezpieczeństwa wątków

```

# Definicja metody Module.synchronize_method, która tworzy łańcuchy aliasów metod egzemplarza,
# dzięki czemu są one synchronizowane na egzemplarzu przed wykonaniem.
class Module
  # Metoda pomocnicza dla tworzenia łańcucha aliasów.
  # Mając podaną nazwę metody (jako łańcuch lub symbol) i prefiks, tworzy
  # unikatowy alias dla tej metody i zwraca nazwę tego aliasu jako symbol.
  # Wszystkie znaki interpunkcyjne w oryginalnej nazwie metody
  # są konwertowane na liczby, dzięki czemu można tworzyć aliasy operatorów.
  def create_alias(original, prefix="alias")
    # Przykleja prefiks do oryginalnej nazwy i konwertuje znaki interpunkcyjne.
    aka = "#{prefix}_#{original}"

```

```

aka.gsub!(/([\=\|\&\+\-\*\|\^\!\?\~\%\<\>\[ \]])/) {
    num = $1[0]                                # Znak Ruby 1.8 -> porządkowy.
    num = num.ord if num.is_a? String           # Znak Ruby 1.9 -> porządkowy.
    '_' + num.to_s
}

# Dodawanie znaków podkreślenia, aż powstanie nazwa, która nie jest jeszcze używana.
aka += "_" while method_defined? aka or private_method_defined? aka
aka = aka.to_sym                            # Konwersja nazwy aliasu na symbol.
alias_method aka, original                 # Rzeczywiste utworzenie aliasu.
aka                                         # Zwrócenie nazwy aliasu.

end
# Dodanie metody do łańcucha aliasów, aby umożliwić synchronizację.
def synchronize_method(m)
    # Najpierw trzeba utworzyć alias dla niesynchronizowanej wersji metody.
    aka = create_alias(m, "unsync")
    # Teraz należy przedefiniować oryginal, aby wywoływał alias w synchronizowanym bloku.
    # Chcesz, aby zdefiniowana metoda przyjmowała bloki, a więc
    # nie możesz użyć metody define_method, konieczne jest wykonanie łańcucha za pomocą
    # metody class_eval. Wszystko pomiędzy znakami %Q{ i pasującym znakiem }
    # jest łańcuchem w podwójnych cudzysłowach, nie blokiem.
    class_eval %Q{
        def #{m}(*args, &block)
            synchronize(self) { #{aka}(*args, &block) }
        end
    }
end
# Niniejszej globalnej synchronizowanej metody można teraz używać na trzy różne sposoby.
def synchronized(*args)
    # Przypadek 1.: z jednym argumentem i blokiem, synchronizacja na obiekcie
    # i wykonanie bloku.
    if args.size == 1 && block_given?
        args[0].mutex.synchronize { yield }
    # Przypadek 2.: z jednym argumentem niebędącym symbolem i bez bloku.
    # Zwraca obiekt klasy SynchronizedObject.
    elsif args.size == 1 and not args[0].is_a? Symbol and not block_given?
        SynchronizedObject.new(args[0])
    # Przypadek 3.: w przypadku wywołania na rzecz modułu bez bloku wyznaczone metody łączone są w łańcuch
    # w celu umożliwienia synchronizacji. Lub, jeśli nie ma żadnych argumentów,
    # do łańcucha zostaje dodana kolejna zdefiniowana metoda.
    elsif self.is_a? Module and not block_given?
        if (args.size > 0) # Synchronizacja wyznaczonych metod.
            args.each { |m| self.synchronize_method(m) }
        else
            # Jeśli nie zostaną określone żadne metody, synchronizacji podlega kolejna zdefiniowana metoda.
            eigenclass = class<<self; self; end
            eigenclass.class_eval do # Użycie klasy eigenclass do zdefiniowania metod klasowych.
                # Definicja metody method_added powiadająccej o zdefiniowaniu kolejnej metody.
                define_method :method_added do |name|
                    # Najpierw trzeba usunąć tę metodę zwrotną.
                    eigenclass.class_eval { remove_method :method_added }
                    # Następnie synchronizujesz metodę, która właśnie została dodana.
                    self.synchronize_method name
                end
            end
        end
    # Przypadek 4.: każde inne wywołanie jest błędem.
    else
        raise ArgumentError, "Nieprawidłowe argumenty metody synchronize()"
    end
end

```

8.11.3. Tworzenie łańcuchów metod na potrzeby śledzenia

Program na listingu 8.10 jest inną wersją programu z listingu 8.4, która śledzi wyznaczone metody obiektu. Na listingu 8.4 użyto delegacji i metody `method_missing` do zdefiniowania metody `Object.trace` zwracającej śledzony obiekt opakowujący. Niniejsza wersja używa techniki tworzenia łańcuchów do modyfikowania metod obiektu na miejscu. Definiuje metody `trace!` i `untrace!` służące do wstawiania i usuwania z łańcucha wyznaczonych metod obiektu.

Ciekawą rzeczą dotyczącą tego programu jest to, że tworzy łańcuchy w inny sposób niż program na listingu 9.9. Tutaj definiowane są metody singletonowe dla obiektu, a dołączanie do definicji oryginalnej metody egzemplarza odbywa się za pomocą metody `super` użytej we-wnętrz metody singletonowej. Nie są tworzone żadne aliasy metod.

Listing 8.10. Tworzenie łańcuchów przy użyciu metod singletonowych na potrzeby śledzenia

```
# Definicja metod egzemplarza trace! i untrace! dla wszystkich obiektów.
# Metoda trace! „tworzy łańcuch” wyznaczonych metod, definiując metody singletonowe,
# które dodają funkcjonalność śledzenia, a następnie wywołując oryginal za pomocą metody super.
# Metoda untrace! usuwa te metody singletonowe, aby pozbyć się funkcjonalności śledzenia.
class Object
  # Śledzi wyznaczone metody, wysyłając dane do strumienia STDERR.
  def trace!(*methods)
    @_traced = @_traced || []
    # Zapamiętuje zbiór śledzonych metod.
    # Jeśli nie zostały wyznaczone żadne metody, zostaną wzięte pod uwagę wszystkie metody publiczne
    # zdefiniowane bezpośrednio (nieodziedziczone) w klasie tego obiektu.
    methods = public_methods(false) if methods.size == 0
    methods.map! { |m| m.to_sym } # Konwertuje wszystkie łańcuchy na symbole.
    methods -= @_traced          # Usuwa metody, które są już śledzone.
    return if methods.empty?     # Kończy działanie wcześniej, jeśli nie ma nic do zrobienia.
    @_traced |= methods          # Dodaje metody do zbioru metod śledzonych.
    # Śledzi fakt rozpoczęcia śledzenia tych metod.
    STDERR << "Śledzenie #{methods.join(', ')} w #{object_id}\n"
    # Metody singletonowe są zdefiniowane w klasie eigenclass.
    eigenclass = class << self; self; end
    methods.each do |m|
      # Dla każdej metody m.
      # Definicja śledzonej wersji singletonowej metody m.
      # Wysyła informacje śledzenia i wywołuje za pomocą metody super
      # metodę egzemplarza, która śledzi.
      # Chcesz, aby definiowane metody mogły przyjmować bloki, a więc
      # nie możesz użyć metody define_method, tylko musisz w zamian wykonać łańcuch.
      # Zauważ, że wszystko, co znajduje się pomiędzy znakami %Q{ i pasującym znakiem } jest
      # łańcuchem w podwójnych cudzysłowach, nie blokiem. Zauważ także, że są
      # dwa poziomy interpolacji łańcuchów. #{} jest interpolowany, gdy
      # definiowana jest metoda singletonowa. łańcuch #{ } jest interpolowany, gdy
      # metoda ta jest wywoływana.
      eigenclass.class_eval %Q{
        def #{m}(*args, &block)
          begin
            STDERR << "Wchodzenie: #{m}(\#{args.join(', ')})\n"
            result = super
            STDERR << "Wychodzenie: #{m} z wartością \#{result}\n"
            result
          rescue
            STDERR << "Anulowanie: #{m}: \${!.class}: \${!.message}"
            raise
          end
        end
      }
    end
  end
end
```

```

# Wyłącza śledzenie wyznaczonych metod lub wszystkich śledzonych metod.
def untrace!(*methods)
  if methods.size == 0      # Jeśli nie została podana żadna metoda,
    methods = @_traced      # wszystkie aktualnie śledzone metody przestaną być śledzone.
    STDERR << "Wyłączanie śledzenia wszystkich metod obiektu #{object_id}\n"
  else                      # W przeciwnym przypadku wyłącza śledzenie.
    methods.map! { |m| m.to_sym } # Konwertuje lańcuchy na symbole.
    methods &= @_traced        # wszystkich wyznaczonych metod, które są śledzone.
    STDERR << "Wyłączanie śledzenia #{methods.join(' ', '')} w #{object_id}\n"
  end
  @_traced -= methods       # Usunięcie tych metod ze zbioru metod śledzonych.
  # Usuwa śledzone metody singletonowe z klasy eigenclass.
  # Zauważ, że wykonyujesz tu za pomocą metody class eval blok kodu, nie lańcuch.
  (class << self; self; end).class_eval do
    methods.each do |m|
      remove_method m      # Metoda undef_method nie działałaby prawidłowo.
    end
  end
  # Jeśli żadne więcej metody nie są śledzone, usuwasz zmienną egzemplarza.
  if @_traced.empty?
    remove_instance_variable : @_traced
  end
end

```

8.12. Języki do wyspecjalizowanych zastosowań

Celem metaprogramowania w języku Ruby jest często utworzenie **języka do wyspecjalizowanych zastosowań**, zwanego potocznie DSL (and. *Domain-Specific Language*). DSL jest rozszerzeniem składni języka Ruby (z metodami wyglądającymi jak słowa kluczowe) lub API pozwalającym bardziej naturalnie zaprezentować dane lub rozwiązać jakiś problem. W prezentowanych przykładach kodu zajmujemy się problemem wysyłania danych na wyjście w formacie XML. Do jego rozwiązania zdefiniowane zostaną dwa języki DSL — jeden prosty i jeden bardziej skomplikowany².

8.12.1. Proste wysyłanie danych w formacie XML przy użyciu metody method_missing

Zaczniemy od prostej klasy o nazwie XML generującej dane w formacie XML. Poniżej znajduje się przykład użycia tej klasy:

```

pagetitle = "Strona testowa metody XML.generate"
XML.generate(STDOUT) do
  html do
    head do
      title { pagetitle }
      comment "To jest test."
    end
    body do
      h1(:style => "font-family:sans-serif") { pagetitle }
      ul :type=>"square" do
        li { Time.now }
        li { RUBY_VERSION }
      end
    end
  end
end

```

² Pełne rozwiązanie tego problemu można znaleźć w API Builder Jima Weiricha dostępnym pod adresem <http://builder.rubyforge.org>.

```

    end
  end
end
end

```

Niniejszy kod nie wygląda jak XML i tylko trochę przypomina kod Ruby. Oto generowane przez niego dane (niektóre złamania wiersza zostały dodane dla klarowności):

```

<html><head>
<title>Strona testowa metody XML.generate</title>
<!-- To jest test -->
</head><body>
<h1 style='font-family:sans-serif'>Strona testowa metody XML.generate</h1>
<ul type='square'>
<li>2007-08-19 16:19:58 -0700</li>
<li>1.9.0</li>
</ul></body></html>

```

W implementacji tej klasy i składni generującej dane w formacie XML zostały wykorzystane:

- struktura blokowa języka Ruby,
- wywołania metod z opcjonalnymi nawiasami,
- składnia do przekazywania literałów haszowych do metod bez klamer,
- metoda `missing_method`.

Listing 8.11 przedstawia implementację tego prostego języka DSL.

Listing 8.11. Prosty język DSL do generowania danych w formacie XML

```

class XML
  # Utwórz egzemplarz tej klasy, określając strumień lub obiekt do
  # przechowywania danych wyjściowych. Może to być dowolny obiekt odpowiadający na <<(String).
  def initialize(out)
    @out = out # Zapamiętuje, gdzie mają zostać wysłane dane wyjściowe.
  end
  # Wysyła wyznaczony obiekt jako CDATA, zwraca wartość nil.
  def content(text)
    @out << text.to_s
    nil
  end
  # Wysyła wyznaczony obiekt jako komentarz, zwraca wartość nil.
  def comment(text)
    @out << "<!-- #{text} -->"
    nil
  end
  # Wysyła znacznik o określonej nazwie i z określonymi atrybutami.
  # Jeśli jest blok, zostanie wywołany, aby wysłać lub zwrócić treść.
  # Zwraca wartość nil.
  def tag(tagname, attributes={})
    # Wysłanie nazwy znacznika.
    @out << "<#{tagname}>"
    # Wysłanie atrybutów.
    attributes.each { |attr,value| @out << " #{attr}='#{value}' " }
    if block_given?
      # Ten blok zawiera treść.
      @out << '>' # Zakończenie znacznika otwierającego.
      content = yield # Wywołanie bloku, aby wysłać lub zwrócił treść.
      if content
        @out << content.to_s # Jeśli została zwrócona jakaś treść,
      end # zostanie wysłana jako łańcuch.
      @out << "</#{tagname}>" # Zamknięcie znacznika.
    end
  end
end

```

```

else
    # W przeciwnym przypadku jest to znacznik pusty, a więc zostaje tylko zamknięty.
    @out << '/'
end
nil # Znaczniki wysyłają się same, a więc nie zwracają żadnej treści.
end
# Poniższy kod odpowiada za zamianę tej klasy w język DSL.
# Po pierwsze: każda nieznana metoda jest traktowana jako nazwa znacznika.
alias method_missing tag
# Po drugie: uruchamia blok w nowym egzemplarzu klasy.
def self.generate(out, &block)
    XML.new(out).instance_eval(&block)
end
end

```

8.12.2. Generowanie danych XML z walidacją i generowaniem metod

Zaprezentowana na listingu 8.11 klasa XML jest przydatna do generowania dobrze sformułowanego kodu XML, ale nie sprawdza, czy dane te są poprawne względem jakiejś konkretnej gramatyki XML. Na listingu 8.12 przedstawiony jest program wykonujący podstawową walidację (nie jest to jednak nawet połowa tego, co jest potrzebne do zapewnienia pełnej poprawności — taki program byłby znacznie dłuższy). Listing ten w rzeczywistości zawiera dwa języki DSL w jednym. Pierwszy z nich to język do definiowania gramatyki XML — zestaw znaczników i ich atrybutów. Można go użyć następująco:

```

class HTMLForm < XMLGrammar
  element :form, :action => REQ,
           :method => "GET",
           :enctype => "application/x-www-form-urlencoded",
           :name => OPT
  element :input, :type => "text", :name => OPT, :value => OPT,
           :maxlength => OPT, :size => OPT, :src => OPT,
           :checked => BOOL, :disabled => BOOL, :readonly => BOOL
  element :textarea, :rows => REQ, :cols => REQ, :name => OPT,
           :disabled => BOOL, :readonly => BOOL
  element :button, :name => OPT, :value => OPT,
           :type => "submit", :disabled => OPT
end

```

Ten pierwszy język DSL jest zdefiniowany przez metodę klasową `XMLGrammar.element`. Aby go użyć, należy utworzyć nową klasę jako podklasę klasy `XMLGrammar`. Metoda `element` jako pierwszy argument przyjmuje nazwę znacznika oraz tablicę asocjacyjną dozwolonych atrybutów jako drugi argument. Kluczami w tej tablicy są nazwy atrybutów. Wartościami odpowiadającymi tym nazwom mogą być domyślne wartości tych atrybutów, stała `REQ` oznaczająca atrybuty wymagane lub stała `OPT` oznaczająca atrybuty opcjonalne. Wywołanie metody `element` powoduje wygenerowanie metody z podaną nazwą w definiowanej podklasie.

Definiowana przez programistę podklaśa klasy `XMLGrammar` jest drugim językiem DSL, za pomocą którego można wygenerować dane XML poprawne względem określonych przez programistę zasad. Klasa `XMLGrammar` nie udostępnia metody `method_missing`, a więc nie ma możliwości użycia znacznika nienależącego do gramatyki. Metoda `tag` wysyłająca na wyjście znaczniki sprawdza atrybuty. Wygenerowanej podklaśi określającej gramatykę używa się podobnie jak klasy `XML` z listingu 8.11:

```

HTMLForm.generate(STDOUT) do
  comment "Prosty formularz HTML."
  form :name => "registration",
    :action => "http://www.example.com/register.cgi" do
      content "Name:"
      input :name => "name"
      content "Address:"
      textarea :name => "address", :rows=>6, :cols=>40 do
        "Podaj adres e-mail."
      end
      button { "Submit" }
    end
  end
end

```

Listing 8.12 przedstawia implementację klasy XMLGrammar.

Listing 8.12. Język DSL do generowania poprawnych danych w formacie XML

```

class XMLGrammar
  # Utwórz egzemplarz tej klasy, określając strumień lub obiekt do
  # przechowywania danych. Obiekt musi odpowiadać na <<(String).
  def initialize(out)
    @out = out # Gdzie mają być wysyłane dane.
  end
  # Wywołuje blok w egzemplarzu, który wysyla dane do wyznaczonego strumienia.
  def self.generate(out, &block)
    new(out).instance_eval(&block)
  end
  # Definiuje dozwolony element (czyli znacznik) w gramatyce.
  # Niniejsza metoda jest językiem DSL specyfikującym gramatykę
  # oraz definiuje metody składające się na język DSL generujący XML.
  def self.element(tagname, attributes={})
    @allowed_attributes ||= {}
    @allowed_attributes[tagname] = attributes
    class_eval %Q{
      def #{tagname}(attributes={}, &block)
        tag(:#{tagname}, attributes,&block)
      end
    }
  end
  # Te staje są używane podczas definiowania wartości atrybutów.
  OPT = :opt      # Dla atrybutów opcjonalnych.
  REQ = :req      # Dla atrybutów wymaganych.
  BOOL = :bool    # Dla atrybutów, których wartośćą jest ich własna nazwa.
  def self.allowed_attributes
    @allowed_attributes
  end
  # Wysyła wyznaczony obiekt jako CDATA, zwraca wartość nil.
  def content(text)
    @out << text.to_s
    nil
  end
  # Wysyła wyznaczony obiekt jako komentarz, zwraca wartość nil.
  def comment(text)
    @out << "<!-- #{text} -->"
    nil
  end
  # Wysyła znacznik o wyznaczonej nazwie i z określonym atrybutem.
  # Jeśli istnieje jakiś blok, zostaje wywołany, aby zwrócił lub wysłał treść.
  # Return nil.
  def tag(tagname, attributes={})
    # Wysyła nazwę znacznika.
    @out << "<#{tagname}>"
    # Sprawdza dozwolone atrybuty tego znacznika.
  end

```

```

allowed = self.class.allowed_attributes[tagname]
# Najpierw należy się upewnić, że każdy z tych atrybutów jest dozwolony.
# Zakładając, że wszystkie są dozwolone, wysyłane są wszystkie wyznaczone.
attributes.each_pair do |key,value|
  raise "nieznany atrybut: #{key}" unless allowed.include?(key)
  @out << " #{key}='#{value}'"
end
# Przegląda dozwolone atrybuty, aby sprawdzić, czy nie zostały
# pominione żadne wymagane atrybuty oraz czy są atrybuty z wartościami
# domyślnymi, które można wysłać na wyjście.
allowed.each_pair do |key,value|
  # Jeśli ten atrybut został już wysłany, nic się nie dzieje.
  next if attributes.has_key? key
  if (value == REQ)
    raise "brak wymaganego atrybutu '#{key}' w znaczniku <#{tagname}>"
  elsif value.is_a? String
    @out << " #{key}='#{value}'"
  end
end
if block_given?
  # Ten blok zawiera treść.
  @out << '>'          # Zakończenie znacznika otwierającego.
  content = yield           # Wywołanie bloku, aby wysłał lub zwrócił treść.
  if content
    @out << content.to_s # będzie wysłana na wyjście jako łańcuch.
  end
  @out << "</#{tagname}>" # Zamknięcie znacznika.
else
  # W przeciwnym przypadku jest to znacznik pusty, a więc należy tylko go zamknąć.
  @out << '/>'
end
nil # Znaczniki wysyłają same siebie, a więc nie zwracają żadnej treści.
end
end

```

Platforma Ruby



Rdzenna biblioteka Ruby definiuje bogate i potężne API, które służy jako platforma do tworzenia programów. Warto poświęcić trochę czasu na opanowanie go, zwłaszcza kluczowych klas String, Array, Hash, Enumerable i IO. Nie znając metod zdefiniowanych w tych klasach, można stracić mnóstwo czasu, opracowując coś, co już dawno zostało opracowane.

Niniejszy rozdział stanowi dokumentację tych metod. Nie jest to wyczerpujące źródło wiedzy o API Ruby, ale próba zilustrowania za pomocą krótkich przykładów kodu sposobów użycia najważniejszych metod najważniejszych rdzennych klas i modułów oraz kilku najważniejszych klas z biblioteki standardowej. Naszym celem było przedstawienie szerokiego wachlarza dostępnych metod, aby przyszły programista wiedział o ich istnieniu i w razie potrzeby potrafił znaleźć ich dokumentację za pomocą narzędzia ri.

Niniejszy długi rozdział został podzielony na następujące podrozdziały tematyczne:

- łańcuchy i przetwarzanie tekstu,
- wyrażenia regularne,
- liczby i matematyka,
- daty i godziny,
- moduł Enumerable i kolekcje Array, Hash i Set,
- wejście i wyjście oraz pliki,
- sieć,
- wątki i współbieżność.

Prezentowane na początku tego rozdziału przykłady kodu zajmują jeden wiersz i demonstrują zastosowanie pojedynczych metod. Natomiast przykłady w dalszej części rozdziału, w dokumentacji wątków i współbieżności, są dłuższe i demonstrują takie często spotykane zadania, jak tworzenie klienta sieciowego czy współbieżne przetwarzanie elementów kolekcji.

9.1. Łańcuchy

W rozdziale 3. opisaliśmy składnię literałów łańcuchowych oraz operatory konkatenacji (+), dołączania (<<), zwielokrotniania (*) i indeksowania łańcuchów ([]). W tym podrozdziale rozszerzone zostaną zdobyte wcześniej wiadomości poprzez demonstrację metod klasy String posiadających nazwy. Kolejne sekcje niniejszego podrozdziału opisują bardziej szczegółowo poszczególne obszary.

Zaczniemy od metod będących alternatywą dla operatorów opisanych w rozdziale 3.:

```
s = "witaj"
s.concat(" świecie")      # Synonim <<. Metoda modyfikująca. Zwraca nowy obiekt s.
s.insert(5, " kochany")   # To samo co s[5, 0] = "kochany". Zmienia s. Zwraca nowy obiekt s.
s.slice(0, 5)            # To samo co s[0,5]. Zwraca podłańcuch.
s.slice!(5,8)           # Usuwanie. To samo co s[5,8]=""'. Zwraca usunięty podłańcuch.
s.eql?("witaj świecie") # Prawda. To samo co ==.
```

Istnieje kilka metod służących do sprawdzania długości łańcuchów:

```
s.length               # => 13: liczy znaki w Ruby 1.9, bajty w 1.8.
s.size                # => 13: size jest synonimem.
s.bytesize            # => 14: długość w bajtach — tylko w Ruby 1.9.
s.empty?              # => false.
"".empty?             # => true.
```

Klasa `String` zawiera następujące metody do przeszukiwaniałańcuchów i podmieniania ich treści (do niektórych z nich jeszcze wróćmy przy wyrażeniach regularnych):

```
s = "hello"
# Znajdowanie położenia podłańcucha lubłańcucha pasującego do wzorca.
s.index('l')          # => 2: indeks pierwszej litery l włańcuchu.
s.index(?l)           # => 2: działa także z kodami znaków.
s.index(/l+/)         # => 2: działa także z wyrażeniami regularnymi.
s.index('l', 3)        # => 3: indeks pierwszej litery l włańcuchu na lub za pozycją 3.
s.index('Ruby')       # => nil: szukanyłańcuch nie został znalezionej.
s.rindex('l')         # => 3: indeks ostatniego po prawej l włańcuchu.
s.rindex('l', 2)       # => 2: indeks ostatniego po prawej l włańcuchu na lub przed pozycją 2.
# Szukanie przedrostków i przyrostków: od Ruby 1.9.
s.start_with? "hell" # => true. Zauważ, że nazwa to start_with, a nie starts_with.
s.end_with? "bells"  # => false.
# Sprawdzanie obecności podłańcucha.
s.include?("ll")     # => true: „hello” zawiera „ll”.
s.include?(?H)        # => false: „hello” nie zawiera znaku H.
# Dopasowywanie do wzorca za pomocą wyrażeń regularnych.
s =~ /[aeiouy]{2}/    # => nil: w słowie „hello” nie ma dwóch samogłosek po kolej.
s.match(/[aeiouy]/)   { |m| m.to_s } # => "e": zwieracą pierwszą samogłoskę.
# Dzieleniełańcuchów na podłańcuchy wedługłańcucha lub wzorca rozdzielającego.
"to jest to".split    # => ["to", "jest", "to"] domyślny podział według spacji.
"hello".split('l')    # => ["he", "", "o"].
"1, 2,3".split(/\s*/) # => ["1", "2", "3"]: przecinek i opcjonalna spacja.
# Podziałłańcucha na dwie części plus znak rozdzielający. Tylko Ruby 1.9.
# Te metody zawsze zwieracą tablicę zawierającą trzyłańcuchy:
"banana".partition("an") # => ["b", "an", "an"].
"banana".rpartition("an") # => ["ban", "an", "a"]: zaczyna od prawej.
"123b".partition(/\d+/)  # => ["1", "23", "b"] działa także z wyrażeniami regularnymi.
# Wyszukiwanie i zamianianie pierwszego (sub, sub!) lub wszystkich (gsub, gsub!).
# wystąpienie określonegołańcucha lub wzorca.
# Więcej informacji na temat metod sub i gsub podanych jest przy opisie wyrażeń regularnych.
s.sub("l", "L")          # => "heLo": zamienia tylko pierwsze wystąpienie.
s.gsub("l", "L")         # => "heLo": zamienia wszystkie wystąpienia.
s.sub!((().().), '\2\1')  # => "ehlo": dopasowuje i zamienia miejscami dwie pierwsze litery.
s.sub!((().().), "\\\2\\\1") # => "hello": podwójne ukośniki dla podwójnych cudzysłówów.
# Metody sub i gsub mogą także obliczyćłańcuch do wstawienia w bloku.
# Dopasowuje pierwszą literę każdego słowa i zamienia ją na wielką.
"Hello world".gsub(/\b./) { |match| match.upcase } # => "Hello World".
# W Ruby 1.9 można podać tablicę asocjacyjną zawierającą odwzorowania dopasowań na tekst zamiany.
s.gsub(/[aeiouy]/, "a"=>0, "e"=>1, "i"=>2) # => "h1ll"
```

W przedostatnim wierszu powyższego kodu została użyta metoda `upcase` konwertującałańcuchy na wielkie litery. Klasa `String` udostępnia kilka metod do zmieniania wielkości liter (nie udostępnia natomiast metod do sprawdzania wielkości lub kategorii liter):

```
# Metody zmieniające wielkość liter.
s = "hello"      # Metody te działają tylko na znakach ASCII.
s.upcase         # => "HELLO".
s.upcase!        # => "HELLO"; modyfikuje obiekt s na miejscu.
s.downcase        # => "hello".
s.capitalize     # => "Hello": pierwsza litera wielka, reszta małe.
s.capitalize!    # => "Hello": modyfikuje obiekt s na miejscu.
s.swapcase       # => "hELLO": modyfikuje wielkość każdej litery.
# Porównywanie z rozpoznawaniem wielkości liter (tylko litery ASCII).
# Metoda casecmp działa jak <=>, a więc zwraca -1 dla mniejszości, 0 dla równości i +1 dla większości.
"world".casecmp("WORLD") # => 0.
"a".casecmp("B")         # => -1 (<=> zwraca w tym przypadku 1).
```

Klasa `String` udostępnia kilka przydatnych metod dodających i usuwających białe znaki. Większość z nich występuje w modyfikującej (kończącej się znakiem !) i niemodyfikującej wersji:

```

s = "hello\r\n"          # Łąncuch ze znakiem końca wiersza.
s.chomp!                 # => "hello": usunięto jeden znak końca wiersza.
s.chomp                  # => "hello": nie ma żadnego znaku końca wiersza, a więc nic się nie zmienia.
s.chomp!                 # => nil: zwrócenie wartości nil oznacza brak zmian.
s.chomp("o")              # => "hell": usunięto „o” z końca łańcucha.
$/ = ";"                 # Ustawienie globalnego separatora rekordów $/ na średnik.
"hello;".chomp            # => "hello": tym razem metoda chomp usuwa średnik.

# Metoda chop usuwa końcowy znak lub znak końca wiersza (\n, \r lub \r\n).
s = "hello\n"
s.chop!                 # => "hello": usunięto znak końca wiersza. Obiekt s został zmodyfikowany.
s.chop                  # => "hell": został usunięty ostatni znak. Obiekt s nie został zmodyfikowany.
" ".chop                 # => "": nie ma znaków do usunięcia.
" ".chop!                # => nil: nic się nie zmienia.

# Usunięcie wszystkich białych znaków (wliczając znaki \t, \r, \n) z lewej strony, prawej strony lub obu stron.
# Metody strip!, lstrip! i rstrip! modyfikują łańcuch na miejscu.
s = "\t hello \n"        # Białe znaki na początku i na końcu.
s.strip                  # => "hello".
s.lstrip                 # => "hello \n".
s.rstrip                 # => "\t hello".
# Wyrównanie do lewej, wyrównanie do prawej lub wyśrodkowanie łańcucha w polu o szerokości n znaków.
# Nie ma modyfikujących wersji tych metod. Zobacz także metodę printf.
s = "x"
s.ljust(3)               # => "x ".
s.rjust(3)                # => " x".
s.center(3)               # => " x ".
s.center(5, '-')          # => "--x--": jako dopełnienie można stosować nie tylko spację.
s.center(7, '--')         # => "--x--": dozwolone są różne znaki jako dopełnienie.

```

Za pomocą metod each_byte i each_line można iterować łańcuchy bajt po bajcie i wiersz po wierszu. W Ruby 1.8 metoda each jest synonimem metody each_line, a klasa String dołącza moduł Enumerable. Należy unikać metody each i związanych z nią iteratorów, ponieważ w Ruby 1.9 została ona usunięta i klasa String nie dołącza już modułu Enumerable. W Ruby 1.9 (i bibliotece jcode w Ruby 1.8) dodano iterator each_char oraz umożliwiono iterację łańcuchów znak po znaku:

```

s = "A\nB"                # Trzy znaki ASCII w dwóch wierszach.
s.each_byte {|b| print b, " " } # Drukuje „65 10 66”.
s.each_line {|l| print l.chomp} # Drukuję „AB”.
# Sekwencyjna iteracja po znakach jako jednoznakowych łańcuchach.
# Działa w Ruby 1.9 i Ruby 1.8 z biblioteką jcode:
s.each_char { |c| print c, " " } # Drukuję „A \n B”.
# Iteracja przez wszystkie znaki jako jednoznakowe łańcuchy.
# Nie działa w przypadku łańcuchów wielobajtowych w Ruby 1.8.
# Działa (powoli) dla łańcuchów wielobajtowych w Ruby 1.9:
0.upto(s.length-1) { |n| print s[n,1], " " }
# W Ruby 1.9 bajty, wiersze i znaki są aliasami.
s.bytes.to_a                # => [65,10,66]: alias dla each_byte.
s.lines.to_a                 # => ["A\n","B"]: alias dla each_line.
s.chars.to_a                 # => ["A", "\n", "B"] alias dla each_char.

```

Klasa String udostępnia kilka metod do przetwarzania liczb z łańcuchów i konwertowania łańcuchów na symbole:

```

"10".to_i                  # => 10: konwertuje łańcuch na liczbę całkowitą.
"10".to_i(2)                # => 2: argument określa podstawę systemu liczenia: od base-2 do base-36.
"10x".to_i                  # => 10: przystopek niebędący liczbą zostaje zignorowany. To samo dotyczy liczb szesnastkowych
                           # → ósemkowych.
" 10".to_i                 # => 10: wiodące spacje są ignorowane.
"ten".to_i                  # => 0: nie zgłasza wyjątku po podaniu nieprawidłowych danych.
"10".oct                   # => 8: przetwarza łańcuch jako liczbę całkowitą o podstawie 8.
"10".hex                   # => 16: przewarza łańcuch jako liczbę całkowitą szesnastkową.
"0xff".hex                 # => 255: liczby szesnastkowe mogą zaczynać się od prefiku 0x.

```

```

"1.1 dozen".to_f    #=> 1.1: przetwarza wiodącą liczbę zmiennoprzecinkową.
"6.02e23".to_f      #=> 6.02e+23: notacja wykładnicza jest obsługiwana.
"one".to_sym         #=> :one -- konwersjałańcucha na symbol.
"two".intern          #=> :two -- metoda intern jest synonimem metody to_sym.

```

Na zakończenie kilka metod o różnym przeznaczeniu:

```

# Zwiększaniełańcucha:
"a".succ           #=> "b": następca litery „a”. Istnieje też metoda succ!.
"aaaz".next        #=> "aba": metoda next jest synonimem powyższej metody. Istnieje też metoda next!.
"a".upto("e") { |c| print c } # Drukuj abcde. Iterotor upto jest oparty na metodzie succ.

# Odwracaniełańcucha:
"hello".reverse     #=> "olleh". istnieje też metoda reverse!.
# Debugging.
"hello\n".dump       #=> "\"hello\n\"": Stosuje znaki zastępcze dla znaków specjalnych.
"hello\n".inspect     # Działal podobnie jak metoda dump.

# Translacja z jednego zestawu znaków na inny.
"hello".tr("aeiouy", "AEIOUY") #=> "hEllo": zamienia samogłoski na wielkie litery. Istnieje też metoda tr!.
"hello".tr("aeiouy", " ")      #=> "h ll": konwertuje samogłoski na spacje.
"bead".tr_s("aeiouy", " ")     #=> "b d": konwertuje i usuwa duplikaty.

# Sumy kontrolne.
"hello".sum           #=> 532: słaba 16-bitowa suma kontrolna.
"hello".sum(8)         #=> 20: 8-bitowa suma kontrolna zamiast 16-bitowej.
"hello".crypt("ab")    #=> "abl0JrMf6llhw": jednokierunkowa kryptograficzna suma kontrolna.
# Przekazuje dwa znaki alfanumeryczne jako domieszkę.
# Wynik może być zależny od platformy.

# Liczenie liter, usuwanie liter, usuwanie duplikatów.
"hello".count('aeiouy') #=> 2: liczy małe samogłoski.
"hello".delete('aeiouy') #=> "hil": usuwa małe samogłoski. Istnieje też metoda delete!.
"hello".squeeze('a-z')   #=> "helo": usuwa zbiory liter. Istnieje też metoda squeeze!.
# Kiedy zostanie podanych więcej niż jeden argument, zostanie użyte przecięcie zbiorów.
# Argumenty zaczynające się od znaku ^ są negowane.
"hello".count('a-z', '^aeiouy') #=> 3: liczy małe spółgłoski.
"hello".delete('a-z', '^aeiouy') #=> "eo": usuwa małe spółgłoski.

```

9.1.1. Formatowanie tekstu

Jak wiadomo, literalyłańcuchowe w podwójnych cudzysłowach pozwalają na interpolację do wolnych wyrażeń Ruby. Na przykład:

```

n, animal = 2, "myszy"
 "#{n+1} ślepe #{animal}" #=> 3 ślepe myszy.

```

Składnia interpolacji literalówłańcuchowych została opisana w rozdziale 3. W języku Ruby dostępna jest też inna składnia pozwalająca interpolować wartości dołańcuchów — klasa String udostępnia operator formatujący %, a moduł Kernel globalne metody printf i sprintf. Metody te i operator % działają bardzo podobnie do funkcji printf spopularyzowanej przez język C. Jedną z zalet formatowania w stylu metody printf w stosunku do zwykłej interpolacjiłańcuchów jest możliwość precyzyjnej kontroli szerokości pól, co przydaje się w generowaniu raportów ASCII. Inna zaleta jest taka, że można określić liczbę znaczących cyfr do wyświetlenia w liczbach zmiennoprzecinkowych, co jest przydatne w zastosowaniach naukowych (i czasami finansowych). W końcu formatowanie za pomocą metody printf pozwala na oddzielenie formatowanej wartości odłańcucha, do którego ma ta wartość być interpolowana. To może być przydatne przy internacionalizacji i lokalizacji aplikacji.

Poniżej znajdują się przykłady użycia operatora %. Pełną dokumentację dyrektyw formatowania używanych przez te metody można znaleźć w metodzie Kernel.strftime:

```

# Alternatywy dla zaprezentowanej wyżej interpolacji.
printf('%d ślepe %s', n+1, animal) # Drukuje '3 ślepe myszy', zwraca wartość nil.
sprintf('%d ślepe %s', n+1, animal) #=> '3 ślepe myszy'.
'%d ślepe %s' [n+1, animal] # Jeśli jest więcej niż jeden argument, należy użyć tablicy po prawej stronie.
# Formatowanie liczb.
'%d' % 10          #=> '10': %d — dziesięcne liczby całkowite.
'%x' % 10          #=> 'a': liczby całkowite szesnastkowe.
'%X' % 10          #=> 'A': duże całkowite liczby szesnastkowe.
'%o' % 10          #=> '12': liczby całkowite ósemkowe.
'%f' % 1234.567   #=> '1234.567000': liczby zmiennooprzecinkowe o pełnej długości.
'%e' % 1234.567   #=> '1.234567e+03': wymusza notację wykładniczą.
'%E' % 1234.567   #=> '1.234567e+03': notacja wykładnicza z literą E.
'%g' % 1234.567   #=> '1234.57': sześć znaczących cyfr.
'%g' % 1.23456E12 #=> '1.23456e+12': Używaj %f lub %e w zależności od rzędu wielkości.
# Szerokość pola.
'%5s' % <<<`      # ` <<<`: wyrównanie do prawej w polu o szerokości pięciu znaków.
'%-5s' % >>>`     # >>> `: wyrównanie do lewej w polu o szerokości pięciu znaków.
'%5d' % 123         # '123': pole ma szerokość pięciu znaków.
'%05d' % 123        # '00123': dopełnienie zerami w polu o szerokości pięciu znaków.
# Precyza.
'%.2f' % 123.456   #'123.46': dwie cyfry po przecinku.
'%.2e' % 123.456   #'1.23e+02': dwie cyfry po przecinku = trzy znaczące cyfry.
'%.6e' % 123.456   #'1.234560e+02': zwróć uwagę na dodane zero.
'%.4g' % 123.456   #'123.5': cztery znaczące cyfry.
# Połączenie pól i precyzyji.
'%6.4g' % 123.456 #'123.5': cztery znaczące cyfry w polu o szerokości sześciu znaków.
'%3s' % 'ruby'     #'ruby': argument lańcuchowy przekracza szerokość pola.
'%3.3s' % 'ruby'   #'rub': precyza wymusza obcięcie lańcucha.
# Wiele argumentów do sformatowania.
args = ['Syntax Error', 'test.rb', 20] # Tablica argumentów.
"%s: in '%s' line %d" % args #=> "Syntax Error: in 'test.rb' line 20".
# Te same argumenty interpolowane w innej kolejności! Dobre do internacionalizacji.
"%2$s:%3$d: %1$s" % args #=> "test.rb:20: Syntax Error".

```

9.1.2. Pakowanie i rozpakowywanie łańcuchów binarnych

Łańcuchy w języku Ruby poza danymi tekstowymi mogą przechowywać także dane binarne. Do pracy z binarnymi formatami plików lub binarnymi protokołami sieciowymi mogą być przydatne metody `Array.pack` i `String.unpack`. Pierwsza z tych metod koduje elementy tablicy w postaci łańcucha binarnego. Natomiast metoda `String.unpack` dekoduje łańcuchy binarne, wydobywa z nich wartości i zwraca je w tablicy. Zarówno operacja kodowania, jak i dekodowania odbywają się pod kontrolą łańcucha formatu, w którym litery określają typ danych i kodowanie, a liczby określają liczbę powtórzeń. Sposób tworzenia tych łańcuchów formatu jest bardzo tajemniczy. Pełną listę kodów literowych można znaleźć w dokumentacji metod `Array.pack` i `String.unpack`. Poniżej znajduje się kilka prostych przykładów:

```

a = [1,2,3,4,5,6,7,8,9,10] # Tablica dziesięciu liczb całkowitych.
b = a.pack('i10')           # Pakuje 10 4-bajtowych liczb całkowitych (i) do łańcucha binarnego b.
c = b.unpack('i*')          # Dekoduje wszystkie (*) 4-bajtowe liczby całkowite z łańcucha binarnego b.
c == a                      #=> true.
m = 'witaj świecie'         # Komunikat do zakodowania.
data = [m.size, m]           # Najpierw długość, potem bajty.
template = 'Sa*'             # Liczba typu short bez znaku, dowolna liczba znaków ASCII.
b = data.pack(template)      #=> "\v\000witaj świecie".
b.unpack(template)           #=> [11, "witaj świecie"].

```

9.1.3. Łańcuchy i kodowanie

Metody klasy `String encode, encode! i force_encoding` oraz klasa `Encoding` zostały opisane w podrozdziale 3.2.6. Planując pisanie programów z wykorzystaniem znaków Unicode lub wielobajtowych, warto przeczytać ten podrozdział jeszcze raz.

9.2. Wyrażenia regularne

Wyrażenie regularne (także `regexp` lub `regex`) wyznacza wzorzec tekstowy. W języku Ruby wyrażenia regularne są zaimplementowane w klasie `Regexp`¹. Metody i operatory dopasowujące wzorce są zdefiniowane w klasach `Regexp` i `String`. Tak jak większość języków obsługujących wyrażenia regularne składnia tych wyrażeń w języku Ruby jest bardzo podobna (ale nie identyczna) do składni wyrażeń regularnych w języku Perl 5.

9.2.1. Literały wyrażeń regularnych

Literały wyrażeń regularnych są oddzielane od reszty kodu ukośnikami:

```
/Ruby?/ #Dopasowuje tekst Rub, po którym może opcjonalnie być litera y.
```

Zamykający ukośnik nie jest prawdziwym ogranicznikiem, ponieważ po literale wyrażenia regularnego może znajdować się jeden lub więcej znaków znaczników określających dodatkowe informacje na temat tego, jak ma być dokonywane dopasowywanie. Na przykład:

```
/ruby?/i #Bez rozróżniania wielkich i małych liter: dopasowuje "ruby" lub "RUB" itd.  
/.mu #Dopasowuje znaki Unicode w trybie Multiline.
```

Dostępne znaki modyfikujące zostały wymienione w tabeli 9.1.

Tabela 9.1. Znaki modyfikujące wyrażeń regularnych

Modyfikator	Opis
i	Ignorowanie wielkości liter.
m	Wzorzec będzie dopasowywany do tekstu składającego się z kilku wierszy, a więc znak nowego wiersza będzie traktowany jako zwykły znak — pozwala dopasowywać znaki nowego wiersza.
x	Rozszerzenie składni — pozwala na używanie spacji i komentarzy w wyrażeniu regularnym.
o	Wykonuje interpolację <code>#{ } </code> tylko jeden raz, za pierwszym razem wykonywany jest literal wyrażenia regularnego.
u, e, s, n	Interpretuje wyrażenie regularne jako Unicode (UTF-8), EUC, SJIS lub ASCII. Jeśli żaden z tych modyfikatorów nie zostanie użyty, przyjmowane jest kodowanie źródła dla wyrażenia regularnego.

Tak jak literały łańcuchowe można zaczynać od znaków `%Q`, wyrażenia regularne mogą zaczynać się od znaków `%r`, po których powinien znajdować się wybrany znak ograniczający. Jest to przydatne, gdy wzorzec zawiera dużo znaków ukośnika, które trzeba by było zastąpić znakami unikowymi:

```
%r| / | #Dopasowuje jeden ukośnik, nie potrzeba znaku unikowego.  
%r[</( .*)>]i #Można też używać znaków znaczników w tej składni.
```

¹ Programiści języka JavaScript powinni zauważyc, że w Ruby w nazwie tej klasy jest tylko jedna wielka litera, nie `RegExp`.

W składni wyrażeń regularnych specjalne znaczenie mają znaki (), [], {}, ., ?, +, *, |, ^ i \$. Aby użyć któregoś z tych znaków we wzorcu, należy użyć znaku lewego ukośnika. Aby utworzyć wzorzec zawierający lewy ukośnik, należy ukośnik ten napisać dwa razy:

```
/\\(/ /      # Dopusowuje otwierający i zamykający nawias.  
/\\/ /      # Dopusowuje jeden lewy ukośnik.
```

Literał wyrażeń regularnych zachowują się jak literaly łańcuchowe w podwójnych cudzysłach, a więc mogą zawierać sekwencje specjalne, takie jak \n, \t i (w Ruby 1.9) \u (pełną listę sekwencji specjalnych można znaleźć w tabeli 3.1 w rozdziale 3.):

```
money = /[$\u20AC\u{a3}\u{a5}]/ # Dopusowuje symbol dolara, euro, funta lub jena.
```

Także podobnie jak literaly łańcuchowe w podwójnych cudzysłach, literaly wyrażeń regularnych pozwalają na interpolację dowolnych wyrażeń Ruby za pomocą składni #{}:

```
prefix = ", "  
/#{prefix}\t/    # Dopusowuje przecinek, po którym znajduje się znak ASCII tabulatora.
```

Należy zauważyć, że interpolacja jest wykonywana przed analizą treści wyrażenia regularnego. Oznacza to, że wszystkie znaki specjalne w interpolowanym wyrażeniu stają się częścią składni tego wyrażenia regularnego. Interpolacja jest w normalnych warunkach wykonywana od nowa przy każdej ewaluacji wyrażenia regularnego. Jeśli jednak zostanie użyty modyfikator o, zostanie ona wykonana tylko jeden raz — przy pierwszej analizie składniowej kodu. Działanie modyfikatora o najlepiej zaprezentować na przykładzie:

```
[1,2].map{|x| /\#\{x\}/}    # => [/1/, /2/]  
[1,2].map{|x| /\#\{x\}o/}   # => [/1/, /1/]
```

9.2.2. Metody fabryczne klasy Regexp

Wyrażenia regularne zamiast literałów mogą być tworzone za pomocą metody `Regexp.new` lub jej synonimu `Regexp.compile`:

```
Regexp.new("Ruby?")           # /Ruby?/  
Regexp.new("ruby?", Regexp::IGNORECASE) # /ruby?/i  
Regexp.compile(".", Regexp::MULTILINE, "u") # /.mu
```

Przed przekazaniem łańcucha do konstruktora klasy `Regexp` należy na jego rzecz wywołać metodę `Regexp.escape`, aby zastąpić sekwencjami specjalnymi znaki specjalne:

```
pattern = "[a-z]+"
```

Jedna lub więcej liter.

```
suffix = Regexp.escape("("))
```

Te znaki są traktowane literalnie.

```
r = Regexp.new(pattern + suffix)
```

#/a-z]+|()/#

W Ruby 1.9 (oraz 1.8.7) metoda fabryczna `Regexp.union` tworzy wzorzec będący sumą dowolnej liczby łańcuchów lub obiektów klasy `Regexp` (oznacza to, że tak utworzony wzorzec pasuje do każdego łańcucha pasującego do któregoś ze składających się na niego wzorców). Metoda ta przyjmuje kilka argumentów lub jedną tablicę łańcuchów i wzorców. Technika ta jest przydatna do tworzenia wzorców pasujących do każdego słowa na liście słów. Znaki specjalne w łańcuchach przekazywanych do metody `Regexp.union` są automatycznie zastępowane sekwencjami specjalnymi, w przeciwnieństwie do metod `new` i `compile`:

```
# Dopusowuje dowolną z pięciu nazw języków.  
pattern = Regexp.union("Ruby", "Perl", "Python", /Java(Script)?/) # Dopusowuje pustą parę nawiasów, nawiasów kwadratowych lub klamer. Stosowanie sekwencji specjalnych jest zautomatyzowane.  
Regexp.union("()", "[ ]", "{}")    # => /\(\)|\[ ]\]|{\}/
```

9.2.3. Składnia wyrażeń regularnych

Wiele języków programowania wykorzystuje składnię wyrażeń regularnych spopularyzowaną przez język Perl. Książka ta nie zawiera pełnego opisu tej składni, ale poniższe przykłady kodu demonstrują niektóre wybrane fragmenty gramatyki wyrażeń regularnych. Na końcu tego kursu znajduje się tabela 9.2 zawierająca zestawienie elementów opisywanej składni. Skoncentrowaliśmy się na składni wyrażeń regularnych języka Ruby 1.8, ale pokazaliśmy także niektóre własności dostępne tylko w Ruby 1.9. Bardziej obszerny opis wyrażeń regularnych można znaleźć w książce *Wyrażenia regularne* autorstwa Jeffreyego E. F. Friedla (Helion).

```
# Znaki literalne.  
/ruby/          # Dopasowuje ruby. Większość znaków dopasowuje same siebie.  
/\$/            # Dopasowuje symbol jena. Znaki wielobajtowe są obsługiwane  
# przez Ruby 1.9 i Ruby 1.8.  
  
# Klasyczne znaki.  
/[Rr]uby/       # Dopasowuje Ruby lub ruby.  
/rub[ye]/       # Dopasowuje ruby lub rube.  
/[aeiouy]/      # Dopasowuje każdą pojedynczą małą samogłoskę.  
/[0-9]/          # Dopasowuje każdą cyfrę, to samo co /[0123456789]/.  
/[a-z]/          # Dopasowuje każdą małą literę ASCII.  
/[A-Z]/          # Dopasowuje każdą wielką literę ASCII.  
/[a-zA-Z0-9]/    # Dopasowuje każde z powyższych.  
/[^aeiouy]/     # Dopasowuje wszystko, co nie jest małą samogłoską.  
/[^\0-9]         # Dopasowuje wszystko, co nie jest cyfrą.  
  
# Klasyczne znaki specjalne.  
. /             # Dopasowuje każdy znak z wyjątkiem nowego wiersza.  
./m             # W trybie wielowierszowym znak . dopasowuje nowy wiersz.  
/\d/            # Dopasowuje cyfrę — /[0-9]/.  
/\D/            # Dopasowuje każdy znak niebędący cyfrą — /[^0-9]/.  
/\s/            # Dopasowuje białe znaki — /[\t\r\n\f]/.  
/\S/            # Dopasowuje znaki niebędące białymi — /[^ \t\r\n\f]/.  
/\w/            # Dopasowuje jeden znak słowa — ze zbioru /[A-Za-z0-9_]/.  
/\W/            # Dopasowuje jeden znak niebędący znakiem słowa — należący do zbioru /[A-Za-z0-9_]/.  
  
# Repetition.  
/ruby?/          # Dopasowuje rub lub ruby — litera y jest opcjonalna.  
/ruby*/          # Dopasowuje rub plus 0 lub więcej liter y.  
/ruby+/          # Dopasowuje rub plus 1 lub więcej liter y.  
/\d{3}/          # Dopasowuje dokładnie trzy cyfry.  
/\d{3,}/         # Dopasowuje trzy lub więcej cyfr.  
/\d{3,5}/        # Dopasowuje 3, 4 lub 5 cyfr.  
  
# Powtarzanie niezachlanne — dopasowywanie najmniejszej liczby powtórzeń.  
/^.*/            # Powtarzanie zachlanne — dopasowuje <ruby>perl</>.  
/^.+*/           # Niezachlanne — dopasowuje <ruby> w <ruby>perl</>. # Kolejne niezachlanne — ??, +? i {n,m}?.  
  
# Grupowanie za pomocą nawiasów.  
/\D\d+/          # Bez grupowania — + powtarza \d.  
/(\D\d+)/         # Zgrupowane — + powtarza parę \D\d.  
/([Rr]uby(, )?)*/ # Dopasowuje „Ruby”, „Ruby, ruby, ruby” itd.  
# Odwołania wstecz — ponowne dopasowywanie wcześniej dopasowanych grup.  
/([Rr]uby&\1ails/ # Dopasowuje ruby&rails lub Ruby&Rails.  
/([. ])[^1]*\1/   # Łancuch w cudzysłowach pojedynczych lub podwójnych.  
# \1 dopasowuje to, co dopasowała pierwsza grupa.  
# \2 dopasowuje to, co dopasowała druga grupa itd.  
  
# Nazwane grupy i odwołania wstecz w Ruby 1.9: dopasowanie 4-literowego palindromu.  
/(?<first>\w)(?<second>\w)\k<second>\k<first>/ # Alternatywna składnia.  
# Alternatywne rozwiązania.  
/ruby|rube/       # Dopasowuje ruby lub rube.  
/rub(y|le))/      # Dopasowuje ruby lub rube.  
/ruby(!+|?)/       # Dopasowuje ruby z jednym lub więcej znaków ! albo jednym znakiem ?.  
# Zakotwiczenia — określenie pozycji dopasowania.
```

```

/^Ruby/          # Dopasowuje Ruby na początku łańcucha lub wewnętrznego wiersza.
/Ruby$/          # Dopasowuje Ruby na końcu łańcucha lub wiersza.
/\\ARuby/         # Dopasowuje Ruby na początku łańcucha.
/Ruby\\Z/         # Dopasowuje Ruby na końcu łańcucha.
/\\bRuby\\b/       # Dopasowuje Ruby na granicy słowa.
/\\brub\\B/        # \\B jest granicą ciągu znaków niebędącego całym słowem:
                  # Dopasowuje rub w rube i ruby, ale niesamodzielne.
/Ruby(?!)/        # Dopasowuje Ruby, jeśli znajduje się po nim wykryznik.
/Ruby(?! !)/      # Dopasowuje Ruby, jeśli nie ma po nim wykryznika.

# Specjalna składnia z nawiasami.
/R(?#comment)/   # Dopasowuje R. Cała reszta jest komentarzem.
/R(?i)uby/        # Nie rozróżnia małych i wielkich liter, dopasowując uby.
/R(?i:uby)/       # To samo.
/rub(?:y|le)/     # Grupowanie tylko bez tworzenia odwołania wstecz typu \\1.
# Opcja x dopuszcza komentarze i ignoruje białe znaki.
/               # To nie jest komentarz Ruby. To jest literalna część
                  # wyrażenia regularnego, która jest ignorowana.
R               # Dopasowuje pojedynczą literę R,
(uby)+          # po której znajduje się jeden lub więcej ciągów uby.
\              # Dla nieignorowanych spacji należy używać wstecznego ukośnika.
/x              # Ogranicznik zamkajający. Nie zapomnij o opcji x!

```

Tabela 9.2 zawiera zestawienie reguł składniowych przedstawionych w niniejszym kodzie

Tabela 9.2. Składnia wyrażeń regularnych

Składnia	Dopasowanie
Klasy znaków	
.	Dopasowuje każdy pojedynczy znak z wyjątkiem znaku nowego wiersza. Użycie opcji <code>m</code> pozwala na dopasowanie także znaku nowego wiersza.
[. . .]	Dopasowuje dowolny pojedynczy znak wymieniony w nawiasach kwadratowych.
[^ . . .]	Dopasowuje każdy pojedynczy znak niewymieniony w nawiasach kwadratowych.
\w	Dopasowuje znaki słowa.
\W	Dopasowuje znaki niebędące znakami słowa.
\s	Dopasowuje białe znaki. Równoznaczne z <code>[\t\n\f]</code> .
\S	Dopasowuje wszystkie znaki z wyjątkiem białych.
\d	Dopasowuje cyfry. Równoznaczne z <code>[0-9]</code> .
\D	Dopasowuje znaki niebędące cyframi.
Szeregi, alternatywy, grupy i odwołania	
ab	Dopasowuje wyrażenie a, po którym znajduje się wyrażenie b.
a b	Dopasowuje wyrażenie a lub wyrażenie b.
(re)	Grupowanie — grupuje re w pojedynczą jednostkę syntaktyczną, której można używać z opcjami *, +, ?, itd. Dodatkowo „zapamiętuje” tekst pasujący do wyrażenia regularnego do późniejszego użytku.
(?:re)	Grupuje tak jak (), ale nie zapamiętuje dopasowanego tekstu.
(?<nazwa>re)	Grupuje podwyrażenie i zapamiętuje tekst pasujący do wyrażenia re jak w () oraz nadaje temu podwyrażeniu etykietę nazwa. Ruby 1.9.
(?'nazwa' re)	Podobnie jak powyżej. Pojedyncze cudzysłowy mogą być opcjonalnie stosowane zamiast nawiasów ostrych. Ruby 1.9.
\1...\\9	Dopasowuje ten sam tekst, który pasował do n-tego zgrupowanego podwyrażenia.
\10...	Dopasowuje ten sam tekst, który pasował do n-tego zgrupowanego podwyrażenia, jeśli jest tyle wcześniejszych podwyrażeń. W przeciwnym przypadku dopasowuje znak z określonym kodowaniem ósemkowym.

Tabela 9.2. Składnia wyrażeń regularnych — ciąg dalszy

Składnia	Dopasowanie
\k<nazwa>	Dopasowuje ten sam tekst, który pasował do wyznaczonej grupy.
\g<n>	Dopasowuje grupę n jeszcze raz. n może być nazwą grupy lub numerem grupy. Porównaj opcję \g, która dopasowuje ponownie lub ponownie wykonuje określona grupę, ze zwykłym odwołaniem wstecz próbującym dopasować ten sam tekst, który pasował za pierwszym razem. Ruby 1.9.
Powtarzanie	Domyślnie powtarzanie jest „zachlane” — dopasowywana jest największa możliwa liczba powtórzeń. Aby dopasowywanie było „powściągliwe”, należy po kwantyfikatorze *, +, ? lub { } postawić znak ?. Wtedy zostanie dopasowanych jak najmniej wystąpień przy zezwoleniu na dopasowanie reszty wyrażenia. W Ruby 1.9 po kwantyfikatorze należy postawić znak +, aby wyłączyć nawracanie.
re*	Dopasowuje zero lub więcej wystąpień re.
re+	Dopasowuje jedno lub więcej wystąpień re.
re?	Opcjonalne: dopasowuje zero lub jedno wystąpienie re.
re{n}	Dopasowuje dokładnie n wystąpień re.
re{n, }	Dopasowuje n lub więcej wystąpień re.
re{n, m}	Dopasowuje co najmniej n i najwyżej m wystąpień re.
Zakotwiczenia	Zakotwiczenia nie dopasowują znaków, ale pozycje o zerowej szerokości pomiędzy znakami, „zakotwiczając” dopasowanie w miejscu, w którym panują określone warunki.
^	Dopasowuje początek wiersza.
\$	Dopasowuje koniec wiersza.
\A	Dopasowuje początek łańcucha.
\Z	Dopasowuje koniec łańcucha. Jeśli łańcuch kończy się znakiem nowego wiersza, dopasowanie kończy się przed tym znakiem.
\z	Dopasowuje koniec łańcucha.
\G	Dopasowuje miejsce, w którym skończyło się ostatnie dopasowanie.
\b	Użyte poza nawiasami dopasowuje granice słów. W nawiasach dopasowuje znak <i>backspace</i> (0x08).
\B	Dopasowuje granice ciągów niebędących słowami.
(?=re)	Asercja pozytywnego wyszukiwania w przód — sprawdza, czy znajdujące się dalej znaki pasują do re, ale nie dodaje tych znaków do dopasowanego tekstu.
(?!re)	Asercja negatywnego wyszukiwania w przód — sprawdza, czy znajdujące się dalej znaki nie pasują do re.
(?<=re)	Asercja pozytywnego wyszukiwania w tył — sprawdza, czy znajdujące się wcześniej znaki pasują do re, ale nie dodaje ich do dopasowanego tekstu. Ruby 1.9.
(?<!re)	Negatywna asercja wyszukiwania w tył — sprawdza, czy znajdujące się wcześniej znaki nie pasują do re. Ruby 1.9.
Różne	
(?onflags- offflags)	Nie dopasowuje niczego, ale włącza znaczniki wyznaczone przez onflags i wyłącza znaczniki wyznaczone przez offflags. Te dwa łańcuchy są kombinacjami w dowolnej kolejności liter modyfikujących i, m oraz x. Takie ustawienia znaczników zaczynają działać w miejscu, w którym pojawiają się w wyrażeniu, i działają do końca tego wyrażenia lub do końca grupy w nawiasach, której są częścią, albo aż zostaną przesunięte przez inne wyrażenie ustalające znaczniki.
(?onflags- offflags:x)	Dopasowuje x, stosując wyznaczone znaczniki tylko do tego podwyrażenia. Jest to grupa niezapamiętująca, jak (? : . . .), z dodatkiem znaczników.
(?#...)	Komentarz — wszystko, co znajduje się w nawiasach, jest ignorowane.
(?>re)	Dopasowuje re niezależnie od reszty wyrażenia, nie biorąc pod uwagę, czy to dopasowanie spowoduje, że reszta wyrażenia nie zostanie dopasowana. Przydatne do optymalizacji niektórych skomplikowanych wyrażeń regularnych. Nawiasy nie zapamiętują dopasowanego tekstu.

9.2.4. Dopasowywanie wzorców za pomocą wyrażeń regularnych

Podstawowym operatorem dopasowywania wzorców w języku Ruby jest `=~`. Jeden z jego operandów musi być wyrażeniem regularnym, a drugi łańcuchem (operator ten jest tak samo zaimplementowany w klasie `Regexp`, jak i `String`, dzięki czemu nie ma znaczenia, czy wyrażenie regularne znajduje się po lewej, czy po prawej stronie). Operator `=~` sprawdza swój operand łańcuchowy, aby dowiedzieć się, czy on lub jakiś podłańcuch pasuje do wzorca określonego wyrażeniem regularnym. Jeśli zostanie znalezione dopasowanie, operator ten zwraca indeks wyznaczający miejsce w łańcuchu, w którym zaczyna się pierwsze dopasowanie. W przeciwnym przypadku zwraca wartość `nil`:

```
pattern = /Ruby?/i          # Dopasowuje Rub lub Ruby bez rozróżniania wielkości liter.  
pattern =~ "backrub"        # Zwraca 4.  
"rub ruby" =~ pattern      # 0.  
pattern =~ "r"               # nil.
```

Po użyciu operatora `=~` programista może być zainteresowany innymi informacjami niż tylko pozycja początku dopasowanego tekstu. Po każdym udanym dopasowaniu (niezwracającym wartości `nil`) zmienna globalna `$~` przechowuje obiekt klasy `MatchData` zawierający pełne informacje o dopasowaniu:

```
"hello" =~ /e\w{2}/          # 1: Dopasowanie litery e, po której znajdują się dwa znaki.  
$~.string                   # hello — cały łańcuch.  
$~.to_s                      # ell — część, która pasowała.  
$~.pre_match                 # h — część znajdująca się przed dopasowanym fragmentem.  
$~.post_match                # o — część znajdująca się za dopasowanym fragmentem.
```

Zmienna `$~` to specjalna zmienna lokalna dla wątku i metody. Dwa wątki działające równolegle widzą różne wartości tej zmiennej. Metoda używająca operatora `=~` nie zmienia wartości zmiennej `$~` widzianej przez metodę wywołującą. Więcej informacji na temat zmiennej `$~` i związanych z nią zmiennych globalnych zostanie podana dalej. Alternatywą w programowaniu obiektowym dla tej czarodziejskiej i nieco tajemniczej zmiennej jest metoda `Regexp.last_match`. Wywołanie tej metody bez żadnych argumentów zwraca taką samą wartość jak odwołanie do zmiennej `$~`.

Obiekt klasy `MatchData` ma większe możliwości, jeśli wyrażenie regularne, które zostało dopasowane, zawiera podwyrażenia w nawiasach. W takim przypadku obiekt klasy `MatchData` zawiera informacje o tym, jaki tekst (oraz gdzie się on zaczyna i kończy) został dopasowany przez każde z podwyrażeń:

```
# To jest wzorzec z trzema podwzorcami.  
pattern = /(Ruby|Perl)(\s+)(rocks|sucks)!/  
text = "Ruby\trocks!"          # Tekst pasujący do wzorca.  
pattern =~ text               # => 0: wzorzec dopasował pierwszy znak.  
data = Regexp.last_match       # => Sprawdzenie szczegółowych informacji o dopasowaniu.  
data.size                     # => 4: Obiekty MatchData zachowują się jak tablice.  
data[0]                        # => "Ruby\trocks!": cały dopasowany tekst.  
data[1]                        # => Ruby — tekst pasujący do pierwszego podwzorca.  
data[2]                        # => \t — tekst pasujący do drugiego podwzorca.  
data[3]                        # => rocks — tekst pasujący do trzeciego podwzorca.  
data[1..2]                      # => ["Ruby", "\t"].  
data[1..3]                      # => ["Ruby", "\t", "rocks"].  
data.values_at(1..3)            # => ["Ruby", "rocks"] — tylko wybrane indeksy.  
data.captures                  # => ["Ruby", "\t", "rocks"] — tylko podwzorce.  
Regexp.last_match(3)            # => "rocks": to samo co Regexp.last_match[3].  
# Początek i koniec dopasowania.  
data.begin(0)                  # => 0: indeks początkowy całego dopasowania.
```

```

data.begin(2)          # => 4: indeks początkowy drugiego podwzorca.
data.end(2)            # => 5: indeks końcowy drugiego podwzorca.
data.offset(3)          # => [5,10]: początek i koniec trzeciego podwzorca.

```

W Ruby 1.9, jeśli wzorzec zawiera nazwane zgrupowania, to obiekt klasy `MatchData` uzyskany z takiego wzorca może być używany jak tablica asocjacyjna, w której kluczami są nazwy grup (jako łańcuchy lub symbole). Na przykład:

```

# Tylko Ruby 1.9.
pattern = /(<!lang>Ruby|Perl) (<ver>\d(\.\d+)) (<review>rocks|sucks) !/
if (pattern =~ "Ruby 1.9.1 rocks!")
  $~[:lang]           # => "Ruby".
  $~[:ver]             # => "1.9.1".
  $~["review"]         # => "rocks".
  $~.offset(:ver)      # => [5,10] początek i koniec numeru wersji.
end
# Nazwy grup i rzutowanie nazw grup na numery grup.
pattern.names        # => {"lang", "ver", "review"}.
pattern.named_captures # => {"lang"=>[1], "ver"=>[2], "review"=>[3]}.

```

Nazwane zgrupowania i zmienne lokalne

W Ruby 1.9, jeśli wyrażenie regularne zawierające nazwane zgrupowania pojawia się literalnie po lewej stronie operatora `=~`, nazwy tych zgrupowań są brane za zmienne lokalne i tekst, który pasuje, jest przypisywany do tych zmiennych. Jeśli dopasowanie nie powiedzie się, zmiennym tym przypisywana jest wartość `nil`. Na przykład:

```

# Tylko Ruby 1.9.
if /(<!lang>\w+) (<ver>\d+\.(\\d+)) (<review>\w+)/ =~ "Ruby 1.9 rules!"
  lang    # => "Ruby".
  ver     # => "1.9".
  review  # => "rules".
end

```

Takie czary można oglądać tylko wtedy, gdy wyrażenie regularne występuje literalnie w kodzie. Jeśli wzorzec jest zapisany w zmiennej lub stałej, zostaje zwrócony przez metodę albo znajduje się po prawej stronie, operator `=~` nie wykonuje tego specjalnego rodzaju przypisywania wartości do zmiennych. Jeśli Ruby zostanie wywołany z opcją `-w`, zgłasza ostrzeżenie, gdy operator `=~` nadpisuje już zdefiniowaną zmienną.

Poza operatorem `=~` klasy `Regexp` i `String` udostępniają także metodę `match`. Metoda ta działa podobnie do operatora dopasowania z tym wyjątkiem, że zamiast zwracać indeks, pod którym znaleziono dopasowanie, zwraca obiekt klasy `MatchData` lub wartość `nil`, jeśli nie znaleziono pasującego tekstu. Przykład użycia tej metody:

```

if data = pattern.match(text)  # Lub — data = text.match(pattern).
  handle_match(data)
end

```

W Ruby 1.9 z wywołaniem metody `match` można związać blok kodu. Jeśli nic nie zostanie dopasowane, blok ten jest ignorowany i metoda `match` zwraca wartość `nil`. Jeśli natomiast zostanie znalezione dopasowanie, do bloku przekazywany jest obiekt klasy `MatchData` i metoda `match` zwraca to, co zwróci blok. Dzięki temu w Ruby 1.9 powyższy kod można zapisać krócej:

```

pattern.match(text) { |data| handle_match(data) }

```

Kolejną zmianą wprowadzoną w Ruby 1.9 jest to, że metody `match` przyjmują opcjonalny drugi argument w postaci liczby całkowitej wyznaczającej miejsce, od którego ma zacząć się wyszukiwanie.

9.2.4.1. Zmienne globalne przechowujące dane dopasowań

Język Ruby przyjął składnię wyrażeń regularnych języka Perl i podobnie jak ten język po każdym dopasowaniu tworzy specjalne zmienne globalne. Dla programistów znających język Perl zmienne te mogą być przydatne. Dla tych, którzy języka tego nie znają, mogą się one okazać nieczytelne! Tabela 9.3 zawiera zestawienie tych zmiennych. Zmienne w drugiej kolumnie są aliasami dostępnymi po dodaniu do programu wiersza require 'English'.

Tabela 9.3. Specjalne zmienne globalne wyrażeń regularnych

Zmienna globalna	English	Alternatywa
\$~	\$LAST_MATCH_INFO	Regexp.last_match
\$&	\$MATCH	Regexp.last_match[0]
\$`	\$PREMATCH	Regexp.last_match.pre_match
\$'	\$POSTMATCH	Regexp.last_match.post_match
\$1	none	Regexp.last_match[1]
\$2, etc.	none	Regexp.last_match[2], etc.
\$+	\$LAST_PAREN_MATCH	Regexp.last_match[-1]

Najważniejsza z wszystkich zmiennych w tabeli 9.3 jest zmienna \$. Wszystkie pozostałe są od niej pochodne. Jeśli wartość zmiennej \$. zostanie ustawiona na obiekt klasy MatchData, wartości pozostałych specjalnych zmiennych ulegną zmianie. Pozostałe zmienne globalne są tylko do odczytu, a więc nie można bezpośrednio zmienić ich wartości. Na koniec należy zapamiętać, że zmienna \$. i wszystkie jej pochodne są lokalne wątkowo i metodowo. Oznacza to, że dwa wątki mogą równocześnie wykonywać dopasowania, nie przeszkadzając sobie nawzajem. Oznacza to także, że wartości tych zmiennych widziane przez kod nie zmieniają się, kiedy w kodzie zostanie wywołana metoda wykonująca dopasowywanie wzorca.

9.2.4.2. Dopasowywanie wzorców przy użyciu łańcuchów

Klasa String udostępnia kilka metod przyjmujących jako argumenty obiekty klasy Regexp. Jeśli łańcuch zostanie zindeksowany za pomocą wyrażenia regularnego, zwrócona zostanie ta część tego łańcucha, która pasuje do wzorca. Jeśli po wyrażeniu regularnym znajduje się liczba całkowita, zwracany jest odpowiedni element obiektu MatchData:

```
"ruby123"[/\d+/]          # "123".
"ruby123"[/([a-z]+)(\d+)/,1] # "ruby".
"ruby123"[/([a-z]+)(\d+)/,2] # "123".
```

Metoda slice jest synonimem operatora indeksowania łańcuchów []. Wersja slice! zwraca tę samą wartość co slice, ale jako efekt uboczny usuwa z łańcucha zwrócony podłańcuch:

```
r = "ruby123"
r.slice!(/\d+/) # Zwraca 123, zamienia r na ruby.
```

Metoda split rozbija łańcuch na tablicę podłańcuchów, używając łańcucha lub wyrażenia regularnego jako znaku oddzielającego:

```
s = "jeden, dwa, trzy"
s.split                  # ["jeden," "dwa," "trzy"] — domyślny znak rozdzielający — spacja.
s.split(", ")            # ["jeden", "dwa", "trzy"] — znak rozdzielający wpisany bezpośrednio.
s.split(/\s*,\s*/)        # ["jeden", "dwa", "trzy"] — spacja kolo przecinka jest opcjonalna.
```

Metoda `index` przeszukuje łańcuch w celu znalezienia znaku, podłańcucha lub wzorca i zwraca jego indeks początkowy. Z obiektem klasy `Regexp` jako argumentem działa bardzo podobnie do operatora `=~`, ale przyjmuje także drugi argument wyznaczający pozycję znaku, od którego ma zacząć szukanie. Dzięki temu można znaleźć inne dopasowania niż pierwsze:

```
text = "hello world"
pattern = /l/
first = text.index(pattern)          # 2 — pierwsze dopasowanie zaczyna się na drugim znaku.
n = Regexp.last_match.end(0)         # 3 — koniec pierwszego dopasowania.
second = text.index(pattern, n)      # 3 — wznowienie szukania od tego miejsca.
last = text.rindex(pattern)          # 9 — metoda rindex szuka od końca.
```

9.2.4.3. Wyszukiwanie i podmienianie

Do najważniejszy metod klasy `String`, które wykorzystują wyrażenia regularne, należą: `sub` (podmienianie), `gsub` (globalne podmienianie) i ich modyfikujące odpowiedniki `sub!` i `gsub!`. Wszystkie te metody wykonują operację wyszukiwania i podmieniania przy użyciu wzorca `Regexp`. Metody `sub` i `sub!` podmieniają pierwsze wystąpienie wzorca. Metody `gsub` i `gsub!` podmieniają wszystkie wystąpienia. Metody `sub` i `gsub` zwracają nowy łańcuch, oryginał pozostawiając w nienaruszonym stanie. Metody `sub!` i `gsub!` modyfikują łańcuch, na rzecz którego zostały wywołane. Jeśli łańcuch zostanie zmodyfikowany w jakikolwiek sposób, te modyfikujące metody zwracają tak zmodyfikowany łańcuch. Jeśli nie ma żadnych modyfikacji, zwracają wartość `nil` (dzięki czemu metod tych można używać w instrukcjach `if` i pętlach `while`):

```
phone = gets
phone.sub!(/#.*/ , "")           # Odczytanie numeru telefonicznego.
phone.gsub!(/\\D/ , ' '=>'-')    # Usunięcie komentarzy Ruby.
                                  # Ruby 1.9: usunięcie wszystkiego z wyjątkiem cyfr, dodatkowo zamienia spacje
na myślniki.
```

Niniejsze metody nie wymagają wyrażeń regularnych. Tekst do podmiany można wyznaczyć także za pomocą zwykłego łańcucha:

```
text.gsub!("rails", "Rails")      # Zamienia wszystkie wystąpienia rails na Rails.
```

Wyrażenia regularne są jednak bardziej elastyczne. Aby na przykład zamienić pierwszą literę ciągu `rails` na wielką, nie zmieniając przy tym nic w ciągu `grails`, należy użyć wyrażenia regularnego:

```
text.gsub!(/\brails\b/, "Rails")  # Zamienia pierwszą literę na wielką.
```

Powodem, dla którego metody te zostały opisane w niniejszym podręczniku, jest to, że zastępnikiem nie musi być koniecznie zwykły łańcuch tekstu. (Zastępniki określone w tablicy asocjacyjnej muszą być zwykłymi łańcuchami). Założymy, że potrzebne jest podmienienie łańcucha w zależności od szczegółów znalezionej dopasowania. Metody wyszukiwania i podmieniania najpierw przetwarzają łańcuch podmiany, a dopiero potem wykonują operację zamiany. Jeśli łańcuch zawiera lewy ukośnik z jedną cyfrą, jest ona używana jako indeks w obiekcie `~`, a tekst z obiektu klasy `MatchData` jest używany w miejscu tego ukośnika i cyfry. Jeśli na przykład łańcuch zawiera sekwencję specjalną `\0`, użyty zostaje cały dopasowany tekst. Jeżeli łańcuch zawiera sekwencję `\1`, użyty zostaje tekst pasujący do pierwszego podwyrażenia. Poniższy kod wyszukuje bez rozróżniania wielkich i małych liter słowo `ruby` i umieszcza je w znaczniku pogrubienia HTML, zachowując wielkość liter:

```
text.gsub(/\bruby\b/i, '<b>\0</b>')
```

Należy pamiętać, że przy użyciu łańcucha podmiany w podwójnych cudzysłowach konieczne jest podwojenie ukośnika.

Niektórych może kusić zrobienie tego samego przy użyciu zwykłej interpolacji łańcuchów:

```
text.gsub(/\bruby\b/i, "<b>#${\$&}</b>")
```

To jednak nie działa, ponieważ w takim przypadku interpolacja jest wykonywana na literale łańcuchowym przed przekazaniem go do metody gsub. Dzieje się to przed dopasowaniem wzorca, a więc zmienne typu \$& są jeszcze niezdefiniowane lub zawierają wartości z poprzedniego dopasowania.

W Ruby 1.9 do nazwanych zgrupowań można odwoływać się za pomocą składni nazwanych odwołań wstecznych (ang. *named backreference*) \k:

```
# Usuwa pary cudzysłówów z łańcucha.  
re = /(?:<quote>['"])(?<body>[^'"]*)\k<quote>/  
puts "To sa 'cudzyslowy'".gsub(re, '\k<body>')
```

Łańcuchy podmiany mogą także odnosić się do innego tekstu niż tekst dopasowany przez zgrupowania. Użyj \&, \` , \' i \+, aby zastąpić w wartości \$&, \$` , \$' i \$+.

Poza statycznymi łańcuchami podmiany metody wyszukiwania i podmiany można także wywoływać z blokiem kodu dynamicznie obliczającym łańcuch podmiany. Argumentem tego bloku jest tekst, który pasował do wzorca:

```
# Użycie liter o takiej samej wielkości w nazwach języków programowania.  
text = "RUBY Java perl PyThOn"           # Tekst do zmodyfikowania.  
lang = /ruby|java|perl|python/i          # Wzorzec do dopasowania.  
text.gsub!(lang) { |l| l.capitalize }     # Poprawienie wielkości liter.
```

W bloku kodu można używać zmiennej \$~ i innych związanych z nią zmiennych, które zostały opisane wcześniej w tabeli 9.3:

```
pattern = /(['"])([^'\"]*)\1/    # Łańcuch znaków w pojedynczych lub podwójnych cudzysłowach.  
text.gsub!(pattern) do  
  if ($1 == '')      # Jeśli to jest łańcuch w podwójnych cudzysłowach,  
    ''#$2''          # zostanie zamieniony na pojedyncze cudzysłowy.  
  else               # Jeśli cudzysłowy były pojedyncze,  
    "\#$2\""         # zostaną zamienione na podwójne.  
  end  
end
```

9.2.4.4. Kodowanie wyrażeń regularnych

W Ruby 1.9 obiekty klasy Regexp udostępniają metodę encoding, podobnie jak łańcuchy. Kodowanie wyrażenia regularnego można określić za pomocą modyfikatorów — u oznacza UTF-8, s — SJIS, e — EUC-JP, a n — brak. Można także określić bezpośrednio kodowanie UTF-8, dodając do wyrażenia sekwencję specjalną \u. Jeśli kodowanie nie zostanie określone jawnie, zostanie zastosowane kodowanie źródła. Jeśli jednak wszystkie znaki w wyrażeniu regularnym należą do zestawu ASCII, zostanie użyte kodowanie ASCII, nawet jeżeli kodowaniem źródła jest jakiś nadzbiór ASCII.

Próba dopasowania do siebie wzorca i łańcucha z różnymi kodowaniami powoduje w Ruby 1.9 zgłoszenie wyjątku. Metoda fixed_encoding? zwraca wartość true, jeśli obiekt klasy Regexp ma inne kodowanie niż ASCII. Jeśli metoda ta zwraca wartość false, można ten wzorzec bezpieczne stosować do wszystkich łańcuchów z kodowaniem ASCII lub nadzbiorem ASCII.

9.3. Liczby i matematyka

W rozdziale 3. zostały opisane różne podklasy liczbowe języka Ruby, sposoby pisania literałów liczbowych oraz arytmetyka liczb całkowitych i zmiennoprzecinkowych. W tym podrozdziale rozszerzone zostaną tamte wiadomości przez opisanie API liczbowego i innych klas związanych z obliczeniami matematycznymi.

9.3.1. Metody działające na wartościach liczbowych

Klasa Numeric i jej podklasy udostępniają przydatne predykaty pozwalające sprawdzić klasę lub wartość liczby. Niektóre z tych predykatów działają tylko na liczbach zmiennoprzecinkowych, a inne na liczbach całkowitych:

```
# Predykaty ogólne.  
0.zero?          #=> true (czy to jest zero?).  
1.0.zero?        #=> false.  
0.0.nonzero?    #=> nil (działa jak false).  
1.nonzero?       #=> 1 (działa jak true).  
1.integer?       #=> true.  
1.0.integer?    #=> false.  
1.scalar?         #=> true: nie liczba zespolona. Ruby 1.9.  
1.0.scalar?      #=> true: nie liczba zespolona. Ruby 1.9.  
Complex(1,2).scalar? #=> false: liczba zespolona. W Ruby 1.8 należy dodać wiersz require 'complex'.  
# Predykaty dla liczb całkowitych.  
0.even?          #=> true (Ruby 1.9).  
0.odd?           #=> false.  
# Predykaty dla liczb zmiennoprzecinkowych.  
ZERO, INF, NAN = 0.0, 1.0/0.0, 0.0/0.0 # Stałe do testowania.  
ZERO.finite?     #=> true: czy ta liczba jest skończona?  
INF.finite?      #=> false.  
NAN.finite?      #=> false.  
ZERO.infinite?   #=> nil: czy ta liczba jest nieskończona? Dodatnia czy ujemna?  
INF.infinite?    #=> 1.  
-INF.infinite?   #=> -1.  
NAN.infinite?    #=> nil.  
ZERO.nan?         #=> false: czy ta liczba jest nieliczbą?  
INF.nan?          #=> false.  
NAN.nan?          #=> true.
```

Klasa Numeric i jej klasy pochodne definiują metody do zaokrąglania liczb. Większość z tych metod jest również zdefiniowana w klasie Numeric, dzięki czemu mogą być używane z liczbami dowolnego typu:

```
# Metody zaokrąglające.  
1.1.ceil          #=> 2 — zaokrąglanie w góre — najmniejsza liczba całkowita >= od argumentu.  
-1.1.ceil         #=> -1 — zaokrąglanie w góre — najmniejsza liczba całkowita >= od argumentu.  
1.9.floor         #=> 1 — zaokrąglanie w dół — największa liczba całkowita <= od argumentu.  
-1.9.floor        #=> -2 — zaokrąglanie w dół — największa liczba całkowita <= od argumentu.  
1.1.round         #=> 1 — zaokrąglanie do najbliższej liczby całkowitej.  
0.5.round         #=> 1 — zaokrąglanie w stronę nieskończoności, kiedy w połowie drogi pomiędzy liczbami całkowitymi.  
-0.5.round        #=> -1 — lub zaokrąglanie w kierunku nieskończoności ujemnej.  
1.1.truncate      #=> 1 — odcięcie części ulamkowej — zaokrąglanie w stronę zera.  
-1.1.to_i          #=> -1 — synonim metody truncate.
```

Oto kilka innych ciekawych numerycznych metod i stałych:

```
# Dla każdej wartości klasy Numeric w Ruby 1.9.  
[n.abs, n<=>0]          # Wartość bezwzględna oraz znak.  
[n.abs, n.angle]          # Wielkość oraz kąt (można użyć n.polar).  
[n.numerator, n.denominator] # Licznik oraz mianownik.
```

```
[n.real, n.imag]           # Część rzeczywista oraz urojona.
# Stałe Float: – mogą być zależne od implementacji.
[Float::MAX, Float::MIN]    # => [1.79769313486232e+308, 2.2250738585072e-308].
Float::EPSILON #=> 2.22044604925031e-16 — najmniejsza rozpoznawalna różnica pomiędzy dwiema liczbami
zmiennoprzecinkowymi.
```

9.3.2. Moduł Math

W module Math zdefiniowane są stałe PI i E, metody trygonometryczne i logarytmiczne oraz kilka innych metod o różnym przeznaczeniu. Metody modułu Math są „funkcjami modułu” (zobacz podrozdział 7.5.3), co oznacza, że można je wywoływać poprzez przestrzeń nazw Math lub dołączyć do klasy i wywoływać tak, jakby były funkcjami globalnymi. Kilka przykładów:

```
# Stałe.
Math::PI                      # => 3.14159265358979.
Math::E                        # => 2.71828182845905.

# Pierwiastki.
Math.sqrt(25.0)                # => 5.0 — pierwiastek kwadratowy.
Math.cbrt(27.0)                # => 3.0 — pierwiastek sześcienny; Ruby 1.9 i późniejsze.
27.0***(1.0/3.0)              # => 3.0 — pierwiastek sześcienny obliczony za pomocą operatora **.

# Logarytmy.
Math.log10(100.0)              # => 2.0 — logarytm o podstawie 10.
Math.log(Math::E**3)            # => 3.0 — logarytm naturalny (o podstawie e).
Math.log2(8)                   # => 3.0 — logarytm o podstawie 2. Ruby 1.9 i późniejsze.
Math.log(16, 4)                # => 2.0 — 2nd arg to log() is the base. Ruby 1.9.
Math.exp(2)                    # => 7.38905609893065 — to samo co Math::E**2.

# Trygonometria.
include Math
sin(PI/2)                      # Oszczędza pisanie — pozwala opuścić przedrostek Math.
cos()                           # => 1.0 — sinus. Argument jest podawany w radianach, nie stopniach.
cos(0)                          # => 1.0 — cosinus.
tan(PI/4)                       # => 1.0 — tangens.
asin(1.0)/PI                   # => 0.5 — arcus sinus. Zobacz też metody acos i atan.
sinh(0)                         # => 0.0 — sinus hiperboliczny. Także cosh, tanh.
asinh(1.0)                      # => 0.0 — funkcja odwrotna do sinusa hiperbolicznego. Także acosh, atanh.

# Konwersja współrzędnych kartezjańskich (x,y) na współrzędne biegunkowe (theta, r).
theta = atan2(y,x)             # Kąt pomiędzy osią X a linią (0,0)-(x,y).
r = hypot(x,y)                 # Przeciwny prostokątna — sqrt(x**2 + y**2).

# Różne funkcje.
f,e = frexp(1024.0)            # => {0.5, 11}: rozkład x na [f,e], x = f*2**e.
x = ldexp(f, e)                # => 1024: oblicza x = f*2**e.
erf(0.0)                        # => 0.0: funkcja błędu.
erfc(0.0)                       # => 1.0: 1-erf(x): komplementarna funkcja błędu.
gamma(5)                         # => 24.0: zmiennoprzecinkowa funkcja gamma.
lgamma(100)                     # => {359.134205369575, 1}: logarytm gamma.
```

9.3.3. Arytmetyka liczb dziesiętnych

Klasa BigDecimal z biblioteki standardowej stanowi wartościową alternatywę dla klasy Float, zwłaszcza w obliczeniach finansowych, w których konieczne jest uniknięcie błędu zaokrąglania nieodłącznie towarzyszącemu obliczeniom arytmetycznym na zmiennoprzecinkowych liczbach binarnych (zobacz podrozdział 3.1.4). Obiekty klasy BigDecimal mogą zawierać nieograniczoną liczbę znaczących cyfr i mogą mieć praktycznie nieograniczone rozmiary (obsługiwane są wykładniki większe od jednego miliarda). Ważniejsze jest to, że stosują one arytmetykę dziesiętną, która pozwala na precyzyjne zaokrąglanie. Na przykład:

```
require "bigdecimal"          # Załadowanie biblioteki standardowej.
dime = BigDecimal("0.1")       # Przekazanie lańcucha do konstruktora, nie Float.
4*dime-3*dime == dime         # true przy użyciu klasy BigDecimal, ale false w klasie Float.
# Obliczanie miesięcznych rat kredytu hipotecznego przy użyciu klasy BigDecimal.
```

```

# Zastosowanie trybu zaokrąglania banker's rounding i ograniczenie do 20 cyfr.
BigDecimal.mode(BigDecimal::ROUND_MODE, BigDecimal::ROUND_HALF_EVEN)
BigDecimal.limit(20)
principal = BigDecimal("200000") # Zawsze przekazujełańcuchy do konstruktora.
apr = BigDecimal("6.5") # Rocznastopaooprocentowania.
years = 30 # Okres kredytu w latach.
payments = years*12 # 12 miesięcznych rat w roku.
interest = apr/100/12 # Konwersja rocznej stopy oprocentowania na miesięczną.
x = (interest+1)**payments # Zauważ potęgowanie przy użyciu klasy BigDecimal.
monthly = (principal * interest * x)/(x-1) # Obliczenie raty miesięcznej.
monthly = monthly.round(2) # Zaokrąglenie do dwóch miejsc po przecinku.
monthly = monthly.to_s("f") # Konwersja nałańcuch nadający się do czytania przez człowieka.

```

Więcej na temat API klasy BigDecimal można dowiedzieć się przy użyciu narzędzia ri. Pełną dokumentację można znaleźć w pliku [ext/bigdecimal/bigdecimal_en.html](#).

9.3.4. Liczby zespolone

Do reprezentacji i obliczeń przy użyciu liczb zespolonych (sumy liczby rzeczywistej i urojonej) służy klasa Complex. Jest to wbudowana klasa w Ruby 1.9 i część biblioteki standardowej w Ruby 1.8. Wywołanie require 'complex' (zarówno w Ruby 1.8, jak i w Ruby 1.9) powoduje przeddefiniowane metody modułu Math tak, że umożliwiają pobieranie oraz zwracanie liczb zespolonych. Alternatywnie w Ruby 1.9 można za pomocą require 'cmath' załadować moduł CMath, który definiuje odpowiedniki metod z modułu Math działające na liczbach zespolonych. Oto kilka przykładów:

```

require "complex" # Ruby 1.8, w Ruby 1.9 metody Math działające na liczbach zespolonych.
c = Complex(0.5,-0.2) # => -0.5-0.2i.
Complex.polar(1,Math::PI/2) # => Complex(0.0, 1.0): utworzenie na podstawie współrzędnych biegunkowych.
i = 1.im # => Complex(0, 1): zamiana na liczbę urojoną.
(2.re - 3.5.im).to_s # => "2.5-3.5i": metoda re tylko w Ruby 1.9.
r,i = c.real, c.imag # => [0.5,-0.2]: Część rzeczywista, część urojona.
m,a = c.polar # => [dlugosc, kat]: Tak samo jak [c.abs,c.angle].
d = c.conj # => .5+2i: zmiana znaku części urojonej.
z = "0+0i".to_c # Funkcja konwersji String na Complex - tylko Ruby 1.9.
10.times { z = z*z + c } # Iteracja obliczająca fraktale zbioru Julii.
1.i**2 # => Complex(-1,0): i*i == -1.
x = Math.sin(z) # Moduł 'complex' przeddefiniuje funkcje Math.
require 'cmath' # Ruby 1.9: definicja modułu CMath dla matematyki na liczbach zespolonych.
CMath.sqrt(-1)==Complex::I # => true.

```

9.3.5. Liczby wymierne

Klasa Rational reprezentuje liczby wymierne (iloraz dwóch liczb całkowitych). Klasa ta jest wbudowana w Ruby 1.9, w Ruby 1.8 jest częścią biblioteki standardowej. Definiuje operatory arytmetyczne do działań na liczbach wymiernych. Klasa ta najlepiej działa w połączeniu z biblioteką mathn, która przeddefiniowuje operacje dzielenia liczb całkowitych, aby zawsze zwracały liczby całkowite. Dodatkowo biblioteka mathn unifikuje arytmetykę w języku Ruby, dzięki czemu klasy Integer, Rational i Complex mogą ze sobą współpracować. Dzielenie za pomocą metody quo zwraca liczbę wymierną, gdy oba argumenty są liczbami całkowitymi. Oto kilka przykładów:

```

require "rational" # Załadowanie biblioteki.
penny = Rational(1, 100) # Cent to 1/100 dolara.
require "mathn" # Wymusza, aby dzielenie liczb całkowitych zwracało liczby całkowite.
nickel = "5/100".to_r # Konwersja String na Rational: tylko Ruby 1.9.
dime = 10.quo 100 # => Rational(1,10).

```

```

change = 2*dime + 3*penny      # => Rational(23,100).
change.numerator               # => 23: licznik.
change.denominator              # => 100: mianownik.
change.to_f                      # => 0.23: konwersja na liczbę Float.
(nickel * dime).to_s            # => "1/200".

```

9.3.6. Wektory i macierze

Biblioteka `matrix` definiuje klasy `Matrix` i `Vector` reprezentujące macierze i wektory liczb oraz operatory arytmetyczne do wykonywania na nich obliczeń arytmetycznych. Algebra liniowa nie jest tematem tej książki, ale w poniższym przykładzie kodu użyto klasy `Vector` do reprezentacji dwuwymiarowego punktu oraz obiektów klasy `Matrix` o wymiarach 2x2 do reprezentacji skalowania i obrotu tego punktu:

```

require "matrix"
# Reprezentacja punktu (1,1) jako wektor [1,1].
unit = Vector[1,1]
# Jednostkowa macierz do transformacji.
identity = Matrix.identity(2)    # Macierz 2x2.
identity*unit == unit           # true — brak transformacji.
# Macierz skaluje punkt przez sx,sy.
sx,sy = 2.0, 3.0;
scale = Matrix[[sx,0], [0, sy]]
scale*unit                       # => [2.0, 3.0] — przeskalowany punkt.
# Niniejsza macierz obraca przeciwne do ruchu wskazówek zegara wokół środka układu.
theta = Math::PI/2                # 90 stopni.
rotate = Matrix[[Math.cos(theta), -Math.sin(theta)],
                [Math.sin(theta), Math.cos(theta)]]
rotate*unit                       # [-1.0, 1.0] — Obrót o 90 stopni.
# Dwie transformacje w jednym.
scale * (rotate*unit)             # [-2.0, 3.0].

```

9.3.7. Liczby losowe

W Ruby do generowania liczb losowych służy globalna funkcja `Kernel.rand`. Bez żadnych argumentów zwraca ona pseudolosową liczbę zmiennoprzecinkową większą lub równą 0,0 i mniejszą niż 1,0. Jeśli jako argument zostanie podana liczba całkowita określająca wartość maksymalną, funkcja ta zwraca pseudolosową liczbę całkowitą większą lub równą 0 i mniejszą niż ten argument. Na przykład:

```

rand          # => 0.964395196505186.
rand          # => 0.390523655919935.
rand(100)     # => 81.
rand(100)     # => 32.

```

Jeśli potrzebna jest powtarzalna sekwencja liczb pseudolosowych (najczęściej do celów testowych), należy do generatora liczb losowych podać ziarno o znanej wartości:

```

srand(0)          # Ziarno o znanej wartości.
[rand(100),rand(100)]  # => [44,47] — sekwencja liczb pseudolosowych.
srand(0)          # Wyzerowanie ziarna, aby powtórzyć sekwencję.
[rand(100),rand(100)]  # => [44,47].

```

Aby otrzymać bezpieczne z punktu widzenia kryptografii liczby losowe, należy użyć modułu `SecureRandom`, który jest częścią biblioteki standardowej w Ruby 1.8.7 oraz 1.9.

9.4. Data i godzina

Klasa Time reprezentuje daty i godziny. Jest ona cienką warstwą przykrywającą funkcje daty i godziny udostępniane przez system operacyjny. Dlatego na niektórych platformach klasa ta może nie być w stanie reprezentować dat sprzed 1970 roku lub po roku 2038. Ograniczenia tego nie mają klasy Date i DateTime (niezaprezentowane tutaj) ze standardowej biblioteki date:

```
# Tworzenie obiektów klasy Time.
Time.now          # Zwraca obiekt reprezentujący aktualny czas.
Time.new          # Synonim Time.now.
Time.local(2007, 7, 8)      # July 8, 2007.
Time.local(2007, 7, 8, 9, 10)  # July 8, 2007, 09:10am — czas lokalny.
Time.utc(2007, 7, 8, 9, 10)    # July 8, 2007, 09:10 UTC.
Time.gm(2007, 7, 8, 9, 10, 11) # July 8, 2007, 09:10:11 GMT (to samo co UTC).
# Jedna mikrosekunda przed rozpoczęciem nowego milenium w Londynie.
# Obiekty tego będziemy używać w wielu przykładach poniżej.
t = Time.utc(2000, 12, 31, 23, 59, 59, 999999)
# Składniki obiektu klasy Time.
t.year           # => 2000.
t.month          # => 12 — grudzień.
t.day            # => 31.
t.wday           # => 0 — dzień tygodnia — 0 oznacza niedzielę.
t.yday           # => 366 — dzień roku — 2000 był rokiem przestępym.
t.hour           # => 23 — zegar dwudziestoczterogodzinny.
t.min            # => 59.
t.sec             # => 59.
t.usec           # => 999999 — mikrosekundy, nie milisekundy.
t.zone           # => UTC — nazwa strefy czasowej.
# Zebranie wszystkich składników w tablicy —
# [sec,min,hour,day,month,year,wday,yday,isdst,zone].
# Zauważ brak mikrosekund.
values = t.to_a   # => [59, 59, 23, 31, 12, 2000, 0, 366, false, "UTC"]
# Tego typu tablice można przekazywać do metod Time.local i Time.utc.
values[5] += 1    # Zwiększenie roku.
Time.utc(*values) # => Mon Dec 31 23:59:59 UTC 2001.
# Strefy czasowe i przestawianie czasu.
t.zone           #=> UTC — zwraca nazwę strefy czasowej.
t.utc?           #=> true — t jest w strefie czasowej UTC.
t.utc_offset     #=> 0 — UTC jest przesunięte o 0 sekund względem UTC.
t.localtime      # Konwersja na lokalną strefę czasową. Modyfikuje obiekt klasy Time!
t.zone           #=> PST (lub inną nazwą lokalnej strefy czasowej).
t.utc?           #=> false.
t.utc_offset     #=> -28800 — 8 godzin przed UTC.
t.gmtime         # Konwersja z powrotem na UTC. Kolejny mutator.
t.getlocal       # Zwraca nowy obiekt klasy Time w lokalnej strefie.
t.getutc         # Zwraca nowy obiekt klasy Time w strefie UTC.
t.isdst          #=> false: strefa UTC nie ma przestawienia czasu. Zauważ brak znaku ?.
t.getlocal.isdst #=> false: nie ma przestawienia czasu w zimie.
# Predykaty dotyczące dni tygodnia — Ruby 1.9.
t.sunday?        #=> true.
t.monday?        #=> false.
t.tuesday?       # itd.
# Formatowanie dat i godzin.
t.to_s           #=> Sun Dec 31 23:59:59 UTC 2000 — Ruby 1.8.
t.to_s           #=> 2000-12-31 23:59:59 UTC — Ruby 1.9 używa formatu ISO-8601.
t.ctime          #=> Sun Dec 31 23:59:59 2000 — inny podstawowy format.
# Metoda strftime interpoluje składniki daty i godziny dołańcucha szablonu.
# Formatowanie niezależne od lokalizacji.
t.strftime("%Y-%m-%d %H:%M:%S") #=> 2000-12-31 23:59:59 — format ISO-8601.
t.strftime("%H:%M")              #=> 23:59 — format 24-godzinny.
t.strftime("%I:%M %p")           #=> 11:59 PM — format 12-godzinny.
# Formaty zależne od lokalizacji.
```

```

t.strftime("%A, %B %d")           # => Sunday, December 31.
t.strftime("%a, %b %d %y")         # => Sun, Dec 31 00 — rok w formacie dwucyfrowym.
t.strftime("%x")                  # => 12/31/00 — format zależny od lokalizacji.
t.strftime("%X")                  # => 23:59:59.
t.strftime("%c")                  # To samo co ctime.

# Przetwarzanie dat i godzin.
require 'parsedate'      # Obszerna biblioteka do przetwarzania dat i godzin.
include ParseDate         # Dобавление parsedate() как функции глобальной.
datestring = "2001-01-01"
values = parsedate(datestring) # [2001, 1, 1, nil, nil, nil, nil, nil].
t = Time.local(*values)       # => Mon Jan 01 00:00:00 -0800 2001.
s = t.ctime                   # => Mon Jan 1 00:00:00 2001.
Time.local(*parsedate(s)) == t # => true.
s = "2001-01-01 00:00:00-0500" # Północ w Nowym Jorku.
v = parsedate(s)             # => [2001, 1, 1, 0, 0, 0, "-0500", nil].
t = Time.local(*v)            # Gubi informację o strefie czasowej!

# Działania arytmetyczne na czasie.
now = Time.now                # Aktualny czas.
past = now - 10                # 10 sekund wcześniej. Time - liczba => Time.
future = now + 10               # 10 sekund od teraz. Time + liczba => Time.
future - now                   # => 10 Time - Time => liczba sekund.

# Porównywanie czasu.
past <= future                 # => -1.
past < future                  # => true.
now >= future                  # => false.
now == now                      # => true.

# Metody pomocnicze działające na jednostkach innych niż sekundy.
class Numeric
  # Konwersja odstępów czasu na sekundy.
  def milliseconds; self/1000.0; end
  def seconds; self; end
  def minutes; self*60; end
  def hours; self*60*60; end
  def days; self*60*60*24; end
  def weeks; self*60*60*24*7; end
  # Konwersja sekund na inne odstępy czasu.
  def to_milliseconds; self*1000; end
  def to_seconds; self; end
  def to_minutes; self/60.0; end
  def to_hours; self/(60*60.0); end
  def to_days; self/(60*60*24.0); end
  def to_weeks; self/(60*60*24*7.0); end
end
expires = now + 10.days          # 10 dni od teraz.
expires - now                   # => 864000.0 sekund.
(expires - now).to_hours        # => 240.0 godzin.

# Czas reprezentowany wewnętrznie w sekundach od (zależnie od platformy) epoki.
t = Time.now.to_i                # => 1184036194 sekund od epoki.
Time.at(t)                       # => sekundy od epoki do obiektu Time.
t = Time.now.to_f                 # => 1184036322.90872 — zawiera 908720 mikrosekund.
Time.at(0)                        # => Wed Dec 31 16:00:00 -0800 1969 — epoka w czasie lokalnym.

```

9.5. Kolekcje

Niniejszy podrozdział opisuje klasy **kolekcyjne** języka Ruby. Kolekcją nazywana jest każda klasa reprezentująca zbiór wartości. Najważniejszymi kolekcjami w języku Ruby są klasy `Array` i `Hash`. W bibliotece standardowej dostępna jest też klasa kolekcyjna `Set`. Wszystkie one mają domieszkę modułu `Enumerable`, co oznacza, że metody tego modułu są uniwersalnymi metodami kolekcji.

9.5.1. Obiekty umożliwiające iterację

Moduł `Enumerable` jest domieszką implementującą na bazie iteratora `each` kilka innych przydatnych metod. Opisane poniżej klasy `Array`, `Hash` i `Set` zawierają moduł `Enumerable`, dzięki czemu implementują wszystkie opisane tutaj metody. Inne warte uwagi klasy umożliwiające iterację to `Range` i `IO`. Moduł `Enumerable` został krótko opisany w podrozdziale 5.3.2. Ten rozdział zawiera bardziej szczegółowe informacje na ten temat.

Należy zauważyć, że niektóre klasy umożliwiające iterację posiadają naturalny porządek, do którego stosuje się ich metoda `each`. Na przykład tablice porządkują swoje elementy rosnąco według indeksów. Klasa `Range` ustala porządek rosnący. Obiekty klasy `IO` porządkują linie tekstu w kolejności wczytywania z pliku lub gniazda. W Ruby 1.9 klasy `Hash` i `Set` (która bazuje na klasie `Hash`) porządkują elementy w kolejności ich wstawiania. Natomiast we wcześniejszych wersjach języka Ruby klasy te porządkują swoje elementy w zasadzie w dowolnej kolejności.

Wiele z metod modułu `Enumerable` zwraca przetworzoną wersję kolekcji lub podkolekcję wybranych elementów kolekcji. Z reguły, jeśli metoda modułu `Enumerable` zwraca kolekcję (w odróżnieniu od pojedynczej wartości wybranej z kolekcji), kolekcja ta jest obiektem klasy `Array`. Nie zawsze tak jednak jest. Na przykład klasa `Hash` przeddefiniowuje metodę `reject`, aby zamiast tablicy jednowymiarowej zwracała tablicę asocjacyjną. Bez względu na to, jaka dokładnie będzie wartość zwrotna, jedno jest pewne — kolekcja zwrócona przez metodę z modułu `Enumerable` sama także będzie umożliwiała iterację.

9.5.1.1. Iteracja i konwersja kolekcji

Zgodnie z definicją każdy obiekt umożliwiający iterację musi udostępniać iterator `each`. Moduł `Enumerable` udostępnia prostą wersję tej metody o nazwie `each_with_index`, która zwraca element kolekcji i liczbę całkowitą. W tablicach jednowymiarowych liczba ta jest indeksem tablicy. W obiektach klasy `IO` wyznacza numer wiersza (aczynając od 0). W pozostałych obiektach liczba ta wyznacza ewentualny indeks tablicy, jeśli kolekcje te zostałyby przekonwertowane na tablice:

```
(5..7).each { |x| print x }                                # Drukuję 567.  
(5..7).each_with_index { |x,i| print x,i }    # Drukuję 506172.
```

W Ruby 1.9 moduł `Enumerable` udostępnia metodę o nazwie `cycle`, która powtarza iterację przez elementy kolekcji, dopóki dostarczony przez programistę blok nie przerwie tej iteracji za pomocą instrukcji `break` lub `return` albo nie zgłosi wyjątku. Podczas pierwszego przechodzenia przez obiekt umożliwiający iterację metoda `cycle` zapisuje elementy w tablicy, a następnie iteruje przez tę tablicę. Oznacza to, że po pierwszym przejściu przez kolekcję jej późniejsze modyfikacje nie mają wpływu na działanie metody `cycle`.

Iteratory `each_slice` i `each_cons` tworzą podtablice kolekcji. Aby były dostępne w Ruby 1.8, należał do programu dodać wiersz `require 'enumerator'`, a w Ruby 1.9 (oraz Ruby 1.8.7) wchodzą one w skład rdzennej biblioteki. Metoda `each_slice(n)` iteruje przez wartości w „kawałkach” o długości `n`:

```
(1..10).each_slice(4) { |x| print x } # Drukuję [1,2,3,4]/[5,6,7,8]/[9,10].
```

Metoda `each_cons` jest podobna do metody `each_slice`, ale używa „przesuwanego okienka”, przechodząc przez kolekcję:

```
(1..5).each_cons(3) { |x| print x }      # Drukuję [1,2,3]/[2,3,4]/[3,4,5].
```

Metoda `collect` stosuje do każdego elementu kolekcji blok kodu i zapisuje zwrocone przez ten blok wartości w tablicy. Jej synonimem jest metoda `map` rzutująca elementy kolekcji na elementy tablicy, dla każdego z tych elementów wykonując blok kodu:

```
data = [1,2,3,4]          # Kolekcja umożliwiająca iterację.  
roots = data.collect { |x| Math.sqrt(x) } # Zbiera pierwiastki kwadratowe podanych danych.  
words = %w[witaj świecie]      # Inna kolekcja.  
upper = words.map { |x| x.upcase } # Rzutowanie na wielkie litery.
```

Metoda `zip` przeplata elementy jednej kolekcji umożliwiającą iterację z elementami zera lub więcej innych kolekcji. Zwraca tablicę elementów (jeden z każdej kolekcji) do związanego z jej wywołaniem bloku. Jeśli nie ma żadnego bloku, zwracana jest tablica tablic:

```
(1..3).zip([4,5,6]) { |x| print x.inspect } # Drukuję [1,4][2,5][3,6].  
(1..3).zip([4,5,6],[7,8]) { |x| print x }     # Drukuję 14725836.  
(1..3).zip('a'...'c') { |x,y| print x,y }    # Drukuję 1a2b3c.  
p (1..3).zip('a'...'z')                      # Drukuję [[1,"a"],[2,"b"],[3,"c"]].  
p (1..3).zip('a'...'b')                      # Drukuję [[1,"a"],[2,"b"],[3,nil]].
```

Moduł `Enumerable` udostępnia metodę `to_a` (i jej synonim `entries`) konwertującą kolekcję umożliwiającą iterację na tablicę. Metoda `to_a` została opisana w tym podrozdziale, ponieważ konwersja ta oczywiście wiąże się z iteracją kolekcji. Elementy powstałej w ten sposób tablicy są uporządkowane w takiej kolejności, w jakiej zostały zwrocone przez iterator `each`:

```
(1..3).to_a      # => [1,2,3].  
(1..3).entries    # => [1,2,3].
```

Jeśli do programu zostanie dodany wiersz `require 'set'`, wszystkie obiekty umożliwiające iterację będą udostępniać metodę konwertującą `to_set`. Zbiory zostały opisane w podrozdziale 9.5.4:

```
require 'set'  
(1..3).to_set      # => #<Set: {1, 2, 3}>.
```

9.5.1.2. Enumeratory i iteratory zewnętrzne

Enumeratory należą do klasy `Enumerable::Enumerator`, która udostępnia zaskakującą mało metod jak na taką potężną konstrukcję iteracyjną. Enumeratory są głównie cechą właściwą dla Ruby 1.9 (oraz Ruby 1.8.7), chociaż pewne ich ślady można znaleźć także w Ruby 1.8 po dołączeniu biblioteki `enumerator`. Obiekty klasy `Enumerator` tworzy się za pomocą metody `to_enum` lub jej aliasu `enum_for` albo poprzez wywołanie metody iteracyjnej bez wymaganego przez nią bloku:

```
e = [1..10].to_enum           # Używa metody Range.each.  
e = "test".enum_for(:each_byte) # Używa metody String.each_byte.  
e = "test".each_byte          # Używa metody String.each_byte.
```

Obiekty klasy `Enumerator` umożliwiają iterację za pomocą metody `each` opartej na innym iteratorze jakiegoś innego obiektu. Obiekty klasy `Enumerator`, poza tym że są obiektami pośredniczącymi umożliwiającymi iterację, zachowują się także jak zewnętrzne iteratory. Aby pobrać elementy kolekcji za pomocą zewnętrznego iteratora, wystarczy wielokrotnie wywołać metodę `next`, aż zgłosi wyjątek `StopIteration`. Metoda `Kernel.loop` ratuje wyjątek `StopIteration`. Jeśli metoda `next` zgłosi wyjątek `StopIteration`, kolejne jej wywołanie zazwyczaj rozpoczęta nową iterację, zakładając, że będącą jej podstawą metoda iteracyjna pozwala na powtarzanie iteracji (na przykład iteratory czytające dane z plików nie umożliwiają powtórnych iteracji). Jeśli wielokrotne iteracje są możliwe, zewnętrzny iterator można uruchomić ponownie przed zgłoszeniem wyjątku `StopIteration`, wywołując metodę `rewind`:

```

"Ruby".each_char.max      # => y — metoda iteracyjna enumeratora.
iter = "Ruby".each_char   # Utworzenie enumeratora.
loop { print iter.next }  # Drukuje „Ruby” — iterator zewnętrzny.
print iter.next           # Drukuje „R” — iterator uruchamia się od początku automatycznie.
iter.rewind               # Wymuszenie ponownego rozpoczęcia iteracji.
print iter.next           # Ponownie drukuje „R”.

```

Mając dany enumerator e, można utworzyć nowy enumerator za pomocą wywołania `e.with_index`. Jak sama nazwa wskazuje, ten nowy enumerator zwraca indeks (lub numer iteracji) razem z wartością, którą zwróciłby oryginalny iterator:

```

# Drukuję 0:R\n1:u\n2:b\n3:y\n.
"Ruby".each_char.with_index.each {|c,i| puts "#{i}:#{c}"}

```

Należy pamiętać, że enumeratory jak wszystkie obiekty klasy `Enumerable` definiują metodę `to_splat`: można poprzedzić enumerator znakiem gwiazdki, aby „rozwinąć” go na poszczególne wartości dla wywołania metody lub równoległego przypisania.

9.5.1.3. Sortowanie kolekcji

Jedną z najważniejszych metod modułu `Enumerable` jest metoda `sort`. Konwertuje ona kolekcję umożliwiającą iterację na tablicę i sortuje elementy tej tablicy. Domyslnie sortowanie odbywa się zgodnie z metodą `<=>` sortowanych elementów. Jeśli jednak zostanie dostarczony blok do wywołania, przekazywane są do niego pary elementów i powinien on zwracać wartości `-1`, `0` i `1` oznaczające ich względową kolejność:

```

w = Set['gruszka', 'Burak', 'marchewka'] # Zbiór słów do posortowania.
w.sort                                     # ['Burak','gruszka','marchewka'] — alfabetycznie.
w.sort {|a,b| b<=>a}                      # ['marchewka','gruszka','Burak'] — odwrotnie.
w.sort {|a,b| acasecmp(b)}                  # ['Burak','gruszka','marchewka'] — ignorowanie wielkości liter.
w.sort {|a,b| b.size<=>a.size}             # ['marchewka','gruszka','Burak'] — odwrotnie według długości.

```

Jeśli blok związany z wywołaniem metody `sort` musi wykonać wiele obliczeń, aby wykonać swoje porównywanie, lepiej jest ze względów wydajnościowych użyć metody `sort_by`. Blok związany z metodą `sort_by` zostanie wywołany jeden raz dla każdego elementu w kolekcji i powinien zwrócić numeryczny „klucz sortowania” dla tego elementu. Kolekcja zostanie posortowana rosnąco według tych kluczy. Dzięki temu klucz sortowania jest obliczany tylko jeden raz dla każdego elementu, a nie dwa razy dla każdego porównywania:

```

# Sortowanie bez rozróżniania wielkości liter.
words = ['marchewka', 'Burak', 'gruszka']
words.sort_by {|x| x.downcase}      # => ['Burak', 'gruszka', 'marchewka'].

```

9.5.1.4. Przeszukiwanie kolekcji

Moduł `Enumerable` udostępnia kilka metod do wyszukiwania pojedynczych elementów w kolekcjach. Metoda `include?` i jej synonim `member?` poszukują elementu równego (według operatora `==`) ich argumentowi:

```

primes = Set[2, 3, 5, 7]
primes.include? 2      # => true.
primes.member? 1       # => false.

```

Metoda `find` i jej synonim `detect` wykonują zвязany z ich wywołaniami blok kodu na rzecz każdego elementu kolekcji po kolejno. Jeśli blok ten zwróci wartość inną od `false` lub `nil`, metoda `find` zwraca ten element i kończy iterację. Jeśli blok zawsze zwraca wartość `nil` lub `false`, metoda `find` zwraca wartość `nil`:

```
# Znajduje pierwszą podtablicę zawierającą liczbę 1.
data = [[1,2], [0,1], [7,8]]
data.find { |x| x.include? 1}      # => [1,2].
data.detect { |x| x.include? 3}    # => nil — nie ma takiego elementu.
```

Metoda `find_index` (nowość w Ruby 1.9) działa jak metoda `index` dla tablic: zwraca indeks pasującego elementu albo pierwszego elementu, który jest dopasowany przez blok:

```
data.find_index [0,1]           # => 1 — pasuje drugi element.
data.find_index { |x| x.include? 1} # => 0 — pasuje pierwszy element.
data.find_index { |x| x.include? 3} # => nil — nie ma takiego elementu.
```

Warto pamiętać, że wartość zwrotna metody `find_index` nie jest zbyt przydatna w takich kolekcjach jak tablice asocjacyjne i zbiory niewykorzystujące indeksów liczbowych.

Moduł `Enumerable` udostępnia jeszcze inne metody przeszukujące zwracające kolekcje, a nie pojedyncze elementy. Metody te zostały opisane w kolejnym podrozdziale.

9.5.1.5. Pobieranie fragmentów kolekcji

Metoda `select` pobiera i zwraca elementy kolekcji, dla których związany z wywołaniem blok zwraca wartości inne niż `false` i `nil`. Synonimem tej metody jest metoda `find_all` działająca jak metoda `find`, ale zwracająca tablicę wszystkich znalezionych elementów:

```
(1..8).select { |x| x%2==0}    # => [2,4,6,8] — pobiera liczby parzyste.
(1..8).find_all { |x| x%2==1}  # => [1,3,5,7] — wyszukuje liczby nieparzyste.
```

Metoda `reject` jest przeciwnieństwem metody `select` — w zwracanej tablicy zapisywane są te elementy, dla których blok zwrócił wartość `nil` lub `false`.

```
primes = [2,3,5,7]
primes.reject { |x| x%2==0}  # => [3,5,7] — odrzuca liczby parzyste.
```

Aby móc zarówno odrzucać, jak i pobierać elementy kolekcji, należy użyć metody `partition`. Zwraca ona tablicę zawierającą dwie tablice. Pierwsza z nich zawiera elementy, dla których blok zwrócił wartości odpowiadające `true`, a druga wartości, dla których blok zwrócił wartość `nil` lub `false`:

```
(1..8).partition { |x| x%2==0}  # => [[2, 4, 6, 8], [1, 3, 5, 7]].
```

Metoda `group_by` dostępna w Ruby 1.9 jest uogólnioną wersją metody `partition`. Zamiast traktować blok jako predykat i zwracać dwie grupy, metoda `group_by` pobiera wartość zwrotną bloku i używa jej jako klucza tablicy asocjacyjnej. Rzutuje ten klucz na tablicę zawierającą wszystkie elementy kolekcji, dla których blok zwrócił tę wartość. Na przykład:

```
# Grupuje nazwy języków programowania według pierwszej litery nazwy.
langs = %w[ java perl python ruby ]
groups = langs.group_by { |lang| lang[0] }
groups # => {"j"=>["java"], "p"=>["perl", "python"], "r"=>["ruby"]}.
```

Metoda `grep` zwraca tablicę elementów pasujących do wartości podanej jako argument, dopasowanie określając przy użyciu operatora `==` argumentu. Kiedy argumentem tej metody jest wyrażenie regularne, działa ona jak polecenie wiersza poleceń systemu Unix `grep`. Jeśli z jej wywołaniem jest związany blok kodu, przetwarza on pasujące elementy, tak jakby na rzecz wyników metody `grep` została wywołana metoda `collect` lub `map`:

```
langs = %w[ java perl python ruby ]
langs.grep(/^p/)          # => [perl, python] — zaczynają się od p.
langs.grep(/^p/) { |x| x.capitalize} # => [Perl, Python] — poprawia na wielkie litery.
data = [1, 17, 3.0, 4]
ints = data.grep(Integer)      # => [1, 17, 4] — tylko liczby całkowite.
small = ints.grep(0..9)        # => [1,4] — tylko z przedziału.
```

W Ruby 1.9 do opisanych wyżej metod dodano jeszcze metody `first`, `take`, `drop`, `take_while` i `drop_while`. Metoda `first` zwraca pierwszy element obiektu umożliwiającego iterację lub po podaniu całkowitoliczbowego argumentu `n` tablicę zawierającą `n` pierwszych elementów. Metody `take` i `drop` przyjmują jako argument liczbę całkowitą. Metoda `take` działa tak jak metoda `first` — zwraca tablicę `n` pierwszych elementów obiektu umożliwiającego iterację, na rzecz którego została wywołana. Metoda `drop` działa odwrotnie — zwraca tablicę wszystkich elementów obiektu umożliwiającego iterację z wyjątkiem `n` pierwszych:

```
p (1..5).first(2)      #=> [1,2].  
p (1..5).take(3)       #=> [1,2,3].  
p (1..5).drop(3)       #=> [4,5].
```

Metody `take_while` i `drop_while` zamiast liczby całkowitej przyjmują blok kodu. Metoda `take_while` przekazuje kolejno elementy obiektu umożliwiającego iterację do pierwszego zwrócenia przez ten blok wartości `nil` lub `false`. Następnie zwraca tablicę zawierającą elementy, dla których blok zwrócił wcześniej wartość `true`. Metoda `drop_while` również przekazuje elementy do bloku, aż po raz pierwszy zwróci wartość `false` lub `nil`. Zwraca natomiast tablicę zawierającą element, dla którego blok zwrócił wartość `false`, i wszystkie kolejne elementy:

```
[1,2,3,nil,4].take_while { |x| x }   #=> [1,2,3] — pobiera, aż do zwrócenia wartości nil.  
[nil, 1, 2].drop_while { |x| !x }    #=> [1,2] — odrzuca początkowe wartości nil.
```

Klasa `Array` definiuje własne efektywne wersje powyższych metod, które nie wymagają, by tablice były iterowane za pomocą `each`.

9.5.1.6. Redukowanie kolekcji

Czasami konieczne jest zredukowanie kolekcji umożliwiającej iterację do jednej wartości reprezentującej jakąś własność tej kolekcji. Do tego celu służą metody `min` i `max` zwracające najmniejszy lub największy element kolekcji (przy założeniu, że elementy można porównywać za pomocą metody `<=>`):

```
[10, 100, 1].min      #=> 1.  
['a', 'c', 'b'].max     #=> 'c'.  
[10, 'a', []].min      #=> ArgumentError — tych elementów nie można porównać.
```

Metody `min` i `max` mogą, podobnie jak metoda `sort`, porównywać dwa elementy za pomocą kodu w bloku. W Ruby 1.9 lepiej zamiast tego użyć metod `min_by` i `max_by`:

```
langs = %w[java perl python ruby]    # Która nazwa jest najdłuższa?  
langs.max { |a,b| a.size <=> b.size } #=> python — blok porównuje 2.  
langs.max_by { |word| word.length }   #=> python — tylko Ruby 1.9.
```

W Ruby 1.9 dostępne są też metody `minmax` i `minmax_by` znajdujące zarówno najmniejszą, jak i największą wartość kolekcji oraz zwracające je w dwuelementowej tablicy `[min, max]`:

```
(1..100).minmax          #=> [1,100] min, max jako liczby.  
(1..100).minmax_by { |n| n.to_s } #=> [1,99] min, max jakołańcuchy.
```

Dostępne są też predykaty `any?` i `all?` wykonujące redukcję kolekcji. Stosują one blok predykatyjny do elementów kolekcji. Metoda `all?` zwraca wartość `true`, jeśli predykat jest prawdziwy (nie jest `nil` ani `false`) dla **wszystkich** elementów kolekcji. Metoda `any?` zwraca wartość `true`, jeśli predykat jest prawdziwy dla przynajmniej jednego elementu kolekcji. W Ruby 1.9 jest też metoda `none?` zwracająca wartość `true` tylko wtedy, gdy predykat nigdy nie zwraca wartości typu `true`. Ponadto w Ruby 1.9 dostępna jest też metoda `one?` zwracająca wartość `true` tylko wtedy, gdy predykat zwraca wartość typu `true` dla dokładnie jednego elementu kolekcji. Jeśli metody te zostaną wywołane bez bloków, po prostu testują elementy kolekcji:

```

c = -2..2
c.all? { |x| x>0}      #=> false — nie wszystkie wartości są > 0.
c.any? { |x| x>0}       #=> true — niektóre wartości są > 0.
c.none? { |x| x>2}       #=> true — żadna wartość nie jest > 2.
c.one? { |x| x>0}        #=> false — więcej niż jedna wartość jest > 0.
c.one? { |x| x>2}        #=> false — żadna wartość nie jest > 2.
c.one? { |x| x==2}        #=> true — jedna wartość == 2.
[1, 2, 3].all?           #=> true — żadna wartość nie jest nil ani false.
[nil, false].any?         #=> false — brak wartości typu true.
[].none?                 #=> true — brak wartości niebędących nil ani false.

```

Kolejną nowością w języku Ruby 1.9 jest metoda `count` zwracająca liczbę elementów w kolekcji równej określonej wartości lub liczbie elementów, dla których związany z wywołaniem blok zwraca wartość `true`:

```

a = [1,1,2,3,5,8]
a.count(1)                  #=> 2 — dwa elementy są równe 1.
a.count { |x| x % 2 == 1}    #=> 4 — cztery elementy są nieparzyste.

```

W końcu metoda `inject` jest metodą ogólnego przeznaczenia do redukcji kolekcji. W Ruby 1.9 aliasem tej metody jest metoda `reduce`. Blok związany z wywołaniem metody `inject` wymaga dwóch argumentów. Pierwszy z nich jest wartością akumulacyjną, a drugi elementem z kolekcji. Wartością akumulacyjną dla pierwszej iteracji jest argument przekazany do metody `inject`. Wartość zwrotna bloku po każdej iteracji jest wartością akumulacyjną dla kolejnej iteracji. Wartość zwrócona przez blok po ostatniej iteracji jest wartością zwrotną metody `inject`. Na przykład:

```

# Ile jest liczb ujemnych?
(-2..10).inject(0) { |num, x| x<0 ? num+1 : num }  #=> 2.
# Suma długości słów.
%w[groch z kapusta].inject(0) { |total, word| total + word.length }  #=> 13.

```

Jeśli do metody `inject` nie zostanie przekazany żaden argument, to do pierwszego wywołania bloku zostaną przekazane dwa pierwsze elementy kolekcji (jeśli kolekcja zawiera tylko jeden element, metoda `inject` zwraca ten element). Ten sposób użycia metody `inject` jest przydatny w wielu zastosowaniach:

```

sum = (1..5).inject { |total,x| total + x}  #=> 15.
prod = (1..5).inject { |total,x| total * x} #=> 120.
max = [1,3,2].inject { |m,x| m>x ? m : x}  #=> 3.
[1].inject { |total,x| total + x}             #=> 1 — blok nie jest wywoływany.

```

W Ruby 1.9 zamiast podawać blok, do metody `inject` można przekazać symbol reprezentujący nazwę jakiejś innej metody (lub jakiegoś operatora). Każdy element kolekcji zostanie przekazany do tej wyznaczonej metody wartości akumulacyjnej i jej wynik będzie nową wartością akumulacyjną. Przy wywoływaniu tej metody w ten sposób z symbolem często używany jest synonim `reduce`:

```

sum = (1..5).reduce(:+)                      #=> 15.
prod = (1..5).reduce(:*)                      #=> 120.
letters = ('a'...'e').reduce("-", :concat)    #=> "-abcde".

```

9.5.2. Tablice jednowymiarowe

Tablice jednowymiarowe są prawdopodobnie najczęściej używaną strukturą danych przez programistów języka Ruby. Literały tablicowe i operatory indeksujące zostały opisane w podrozdziale 3.3. Niniejszy podrozdział rozszerza podane wcześniej informacje, demonstrując bogate API klasy `Array`.

9.5.2.1. Tworzenie tablic

Tablicę można utworzyć za pomocą literała tablicowego, metody klasowej `Array.new` lub operatora `Array[]`. Na przykład:

```
[1,2,3]                      # Podstawowy literal tablicowy.  
[]                           # Pusta tablica.  
[]                           # Tablice można modyfikować — ta pusta tablica jest inną.  
%w[a b c]                   # => ['a', 'b', 'c'] — tablica słów.  
Array[1,2,3]                 # => [1,2,3] — to samo co literal tablicowy.  
# Tworzenie tablic za pomocą metody new().  
empty = Array.new             # [] — zwraca nową pustą tablicę.  
nils = Array.new(3)           # [nil, nil, nil] — trzy elementy nil.  
copy = Array.new(nil)         # Robi kopię istniejącej tablicy.  
zeros = Array.new(4, 0)        # [0, 0, 0, 0] — cztery elementy 0.  
count = Array.new(3){|i| i+1}  # [1,2,3] — trzy elementy obliczone przez blok.  
# Należy uważać na powtarzające się obiekty.  
a=Array.new(3, 'a')          # => ['a','a','a'] — trzy referencje do tego samego łańcucha.  
a[0].upcase!                 # Zamiana pierwszego elementu na wielką literę.  
a                         # => ['A','A','A'] — wszystkie te łańcuchy to ten sam łańcuch!  
a=Array.new(3){'b'}           # => ['b','b','b'] — trzy osobne łańcuchy.  
a[0].upcase!;                # Zamiana pierwszego na wielką literę.  
a                         # => ['B','b','b'] — pozostałe łańcuchy pozostają niezmienione.
```

Poza metodami fabrykującymi klasy `Array` w niektórych klasach dostępna jest metoda `to_a` zwracająca tablicę. W szczególności na tablicę za pomocą metody `to_a` można konwertować obiekty umożliwiające iterację, jak klasy `Range` czy `Hash`. Dodatkowo operatory tablicowe, jak `+`, i wiele metod tablicowych, jak `slice`, tworzy i zwraca nowe tablice zamiast modyfikować te, na rzecz których są wywoływanie.

9.5.2.2. Rozmiar tablicy i jej elementy

Poniższy fragment kodu demonstruje, jak sprawdzić długość tablicy oraz kilka sposobów wydobywania z tablicy elementów i podtablic:

```
# Długość tablicy.  
[1,2,3].length               # => 3.  
[].size                       # => 0 — synonim metody length.  
[].empty?                     # => true.  
[nil].empty?                  # => false.  
[1,2,nil].nitems              # => 2 — liczba elementów nie będących nil.  
[1,2,3].nitems { |x| x>2 }   # => 1 — liczba elementów pasujących do bloku (Ruby 1.9 oraz Ruby 1.8.7).  
# Indeksowanie pojedynczych elementów.  
a = %w[a b c d]              # => ['a', 'b', 'c', 'd'].  
a[0]                          # => 'a' — pierwszy element.  
a[-1]                         # => 'd' — ostatni element.  
a[a.size-1]                    # => 'd' — ostatni element.  
a[-a.size]                     # => 'a' — pierwszy element.  
a[5]                          # => nil — nie ma takiego elementu.  
a[-5]                         # => nil — nie ma takiego elementu.  
a.at(2)                        # => 'c' — to samo co [] dla pojedynczego argumentu.  
a.fetch(1)                     # => 'b' — również to samo co [] i at.  
a.fetch(-1)                    # => 'd' — działa z argumentami ujemnymi.  
a.fetch(5)                      # => IndexError! — nie pozwala na wyjście poza granice.  
a.fetch(-5)                     # => IndexError! — nie pozwala na wyjście poza granice.  
a.fetch(5, 0)                   # => 0 — zwraca drugi argument, jeśli wyjdzie poza granice.  
a.fetch(5){|x|x*x}             # => 25 — oblicza wartość, jeśli wyjdzie poza granice.  
a.first                        # => 'a' — pierwszy element.  
a.last                          # => 'd' — ostatni element.  
a.choice                        # Ruby 1.9 — zwraca jeden losowy element.  
# Indeksowanie podtablic.
```

```

a[0, 2]          # => ['a', 'b'] — dwa elementy, zaczynając od 0.
a[0..2]          # => ['a', 'b', 'c'] — elementy z indeksem należącym do zbioru.
a[0...2]          # => ['a', 'b'] — trzy kropki zamiast dwóch.
a[1, 1]          # => ['b'] — pojedynczy element jako tablica.
a[-2, 2]          # => ['c', 'd'] — dwa ostatnie elementy.
a[4, 2]          # => [] — pusta tablica na końcu.
a[5,1]           # => nil — dalej nic nie ma.
a.slice(0..1)    # => ['a', 'b'] — slice to synonim [].
a.first(3)       # => ['a', 'b', 'c'] — trzy pierwsze elementy.
a.last(1)        # => ['d'] — ostatni element jako tablica.

# Wybieranie dowolnych wartości.
a.values_at(0, 2)      # => ['a', 'c'].
a.values_at(4, 3, 2, 1)  # => [nil, 'd', 'c', 'b'].
a.values_at(0, 2..3, -1) # => ['a', 'c', 'd', 'd'].
a.values_at(0..2, 1..3)  # => ['a', 'b', 'c', 'b', 'c', 'd'].

```

9.5.2.3. Modyfikowanie elementów tablicy

Poniższy kod demonstruje sposoby zmiany wybranych elementów tablicy, wstawiania wartości do tablicy, usuwania z niej wartości i zastępowania wartości innymi wartościami:

```

a = [1, 2, 3]          # Tablica początkowa.
# Zmianie wartości elementów.
a[0] = 0                # Modyfikacja istniejącego elementu — a = [0, 2, 3].
a[-1] = 4               # Zmiana ostatniego elementu — a = [0, 2, 4].
a[1] = nil              # Ustawienie drugiego elementu na nil — a = [0, nil, 4].
# Dodawanie elementów do tablicy.
a = [1, 2, 3]          # Tablica początkowa.
a[3] = 4                # Dodanie czwartego elementu do tablicy — a = [1, 2, 3, 4].
a[5] = 6                # Elementy można pomijać — a = [1, 2, 3, 4, nil, 6].
a << 7                 # => [1, 2, 3, 4, nil, 6, 7].
a << 8 << 9            # => [1, 2, 3, 4, nil, 6, 7, 8, 9] — można tworzyć łańcuchy operatorów.
a = [1, 2, 3]          # Nowa krótka tablica początkowa.
a + a                  # => [1, 2, 3, 1, 2, 3] — operator + łączy dwie tablice w nową tablicę.
a.concat([4, 5])        # => [1, 2, 3, 4, 5] — modyfikacja tablicy a. Zauważ brak znaku !.
# Wstawianie elementów za pomocą metody insert.
a = ['a', 'b', 'c']
a.insert(1, 1, 2)       # Tablica a zawiera teraz elementy ['a', 1, 2, 'b', 'c']. Jak a[1, 0] = [1, 2].
# Usuwanie (i zwracanie) wybranych elementów według indeksów.
a = [1, 2, 3, 4, 5, 6]
a.delete_at(4)          # => 5 — a = [1, 2, 3, 4, 6].
a.delete_at(-1)         # => 6 — a = [1, 2, 3, 4].
a.delete_at(4)          # => nil — a pozostaje bez zmian.
# Usuwanie elementów według wartości.
a.delete(4)             # => 4 — a = [1, 2, 3].
a[1] = 1                # a = [1, 1, 3].
a.delete(1)             # => 1 — a = [3] — obie jedynki zostały usunięte.
a = [1, 2, 3]
a.delete_if { |x| x%2==1} # Usunięcie wartości nieparzystych — a = [2].
a.reject! { |x| x%2==0}   # Tak jak delete_if — a = [].

# Usuwanie elementów i podtablic za pomocą metody slice!.
a = [1, 2, 3, 4, 5, 6, 7, 8]
a.slice!(0)              # => 1 — usuwa element 0 — a = [2, 3, 4, 5, 6, 7, 8].
a.slice!(-1, 1)           # => [8] — usuwa podtablicę z końca — a = [2, 3, 4, 5, 6, 7].
a.slice!(2..3)            # => [4, 5] — działa z przedziałami — a = [2, 3, 6, 7].
a.slice!(4, 2)            # => [] — pusta tablica za końcem — a pozostaje bez zmian.
a.slice!(5, 2)            # => nil — a = [2, 3, 6, 7, nil]!
# Zastępowanie podtablic za pomocą metody []=.
# Aby usunąć elementy, należy przypisać pustą tablicę.
# Aby wstawić elementy, należy przypisać do fragmentu o zerowej szerokości.
a = ('a'..'e').to_a       # => ['a', 'b', 'c', 'd', 'e'].
a[0, 2] = ['A', 'B']      # Teraz a = ['A', 'B', 'c', 'd', 'e'].

```

```

a[2...5]=['C', 'D', 'E'] # Teraz a = ['A', 'B', 'C', 'D', 'E'].
a[0,0] = [1,2,3]          # Wstawianie elementów na początek tablicy a.
a[0..2] = []              # Usunięcie tych elementów.
a[-1,1] = 'Z'             # Zastąpienie ostatniego elementu innym.
a[-1,1] = ''               # Dla pojedynczego elementu tablica jest opcjonalna.
a[1,4] = nil              # Ruby 1.9 — teraz a = ['A',nil].
                           # Ruby 1.8 teraz a = ['A'] — nil działa jak [].

# Inne metody.
a = [4,5]
a.replace([1,2,3])        # Teraz a = [1,2,3].
a.fill(0)                 # Teraz a = [0,0,0].
a.fill(nil,1,3)            # Teraz a = [0,nil,nil,nil].
a.fill('a',2..4)           # Teraz a = [0,nil,'a','a','a'].
a[3].upcase!              # Teraz a = [0,nil,A',A',A'].
a.fill(2..4) { 'b' }       # Teraz a = [0,nil,'b','b','b'].
a[3].upcase!
a.compact                 # => [0,'b','B','b'] — kopiowanie z usunięciem wartości nil.
a.compact!                # usuńcie wartości nil na miejscu — teraz a = [0,'b','B','b'].
a.clear                   # Teraz a = [].

```

9.5.2.4. Iteracja, przeszukiwanie i sortowanie tablic

Klasa Array ma domieszkę modułu Enumerable, dzięki czemu dostępne są w niej wszystkie iteratory tego modułu. Ponadto klasa Array definiuje kilka własnych ważnych iteratorów i związanych z nimi metod szukających i sortujących. W Ruby 1.9 oraz 1.8.7 iteratory tablic, gdy są wywoływane bez bloków, zwracają enumerator:

```

a = ['a', 'b', 'c']
a.each {|elt| print elt}          # Podstawowy iterator each drukuje abc.
a.reverse_each {|e| print e}       # Iterator tablicowy drukuje cba.
a.cycle {|e| print e}             # Ruby 1.9 — drukuje "abcabcabc..." w nieskończoność.
a.each_index {|i| print i}         # Iterator tablicowy drukuje 012.
a.each_with_index{|e,i| print e,i} # Enumerable — drukuje a0b1c2.
a.map { |x| x.upcase}             # Enumerable — zwraca ['A','B','C'].
a.map! { |x| x.upcase}            # Tablicowy — modyfikuje na miejscu.
a.collect! { |x| x.downcase!}      # collect! jest synonimem map!.
# Searching methods
a = %w[h e l l o]
a.include?('e')                  # => true.
a.include?('w')                  # => false.
a.index('l')                     # => 2 — indeks pierwszego dopasowania.
a.index('L')                     # => nil — nic nie dopasowano.
a.rindex('l')                    # => 3 — szukanie wstecz.
a.index{|c| c =~ /[aeiou]/}       # => 1 — indeks pierwszej samogłoski. Ruby 1.9 oraz Ruby 1.8.7.
a.rindex{|c| c =~ /[aeiou]/}       # => 4 — indeks ostatniej samogłoski. Ruby 1.9 oraz Ruby 1.8.7.
# Sortowanie.
a.sort    # => %w[h l l o] — skopiowanie tablicy a i posortowanie tej kopii.
a.sort!   # Sortowanie na miejscu — teraz tablica a = ['e', 'h', 'l', 'l', 'o'].
a = [1,2,3,4,5]                  # Nowa tablica do posortowania na liczby parzyste i nieparzyste.
a.sort! { |a,b| a%2 <=> b%2}    # Porównywanie elementów modulo 2.
# Tasowanie elementów — przeciwnieństwo sortowania. Tylko Ruby 1.9.
a = [1,2,3]                      # Tablica uporządkowana.
puts a.shuffle # Tasuje losowo. Np. [3,1,2]. Istnieje też metoda shuffle!.

```

9.5.2.5. Porównywanie tablic

Dwie tablice są sobie równe wtedy i tylko wtedy, gdy mają taką samą liczbę elementów, ich elementy mają takie same wartości i są posortowane w takiej samej kolejności. Metoda == sprawdza równość swoich elementów za pomocą ==, a metoda eql? sprawdza równość swoich elementów, wywołując na nich rzecz metodę eql?. W większości przypadków te dwie metody porównujące zwracają tę samą wartość.

Klasa `Array` nie umożliwia porównywania, ale implementuje operator `<=>` i definiuje porządek dla tablic. Porządek ten jest analogiczny do porządku łańcuchów. Tablice kodów znaków są sortowane w taki sam sposób jak odpowiadające im obiekty klasy `String`. Tablice są porównywane element po elemencie, zaczynając od indeksu 0. Jeśli którakolwiek z par nie jest równa, metoda porównująca tablice zwraca taką samą wartość, jaka została zwrócona przy porównywaniu elementów. Jeśli wszystkie pary elementów są równe i porównywane, tablice mają taką samą długość, uznawane są za równe i metoda `<=>` zwraca wartość 0. W przeciwnym przypadku jedna z tablic jest prefiksem drugiej. W takim przypadku dłuższa tablica jest większa niż tablica krótsza. Należy zauważyć, że pusta tablica `[]` jest prefiksem każdej innej tablicy i jest zawsze mniejsza od tablicy niepustej. Ponadto jeśli jakieś pary elementów nie można porównać (jeśli na przykład jeden element jest liczbą, a drugi łańcuchem), metoda `<=>` zwraca wartość `nil` zamiast `-1`, `0` lub `1`:

```
[1,2] <=> [4,5]      # => -1 ponieważ 1 < 4.
[1,2] <=> [0,0,0]    # => +1 ponieważ 1 > 0.
[1,2] <=> [1,2,3]    # => -1 ponieważ pierwsza tablica jest krótsza.
[1,2] <=> [1,2]      # => 0 — są równe.
[1,2] <=> []          # => +1 — pusta tablica jest zawsze mniejsza od tablicy niepustej.
```

9.5.2.6. Tablice jako stosy i kolejki

Metody `push` i `pop` dodają i usuwają elementy z końca tablicy. Dzięki nim tablic można używać jako stosów typu „ostatni przyszedł, pierwszy wyjdzie”:

```
a = []
a.push(1)      # => [1] — a = [1].
a.push(2,3)    # => [1,2,3] — a = [1,2,3].
a.pop         # => 3 — a = [1,2].
a.pop         # => 2 — a = [1].
a.pop         # => 1 — a = [].
a.pop         # => nil — a nadal = [].
```

Metoda `shift` działa podobnie jak metoda `pop`, ale usuwa i zwraca pierwszy element tablicy zamiast ostatniego. Metoda `unshift` działa podobnie jak metoda `push`, ale dodaje elementy na początku zamiast na końcu tablicy. Za pomocą metod `push` i `shift` można zaimplementować kolejkę typu „pierwszy przyszedł, pierwszy wyjdzie”:

```
a = []
a.push(1)      # => [1] — a = [1].
a.push(2)      # => [1,2] — a = [1,2].
a.shift        # => 1 — a = [2].
a.push(3)      # => [2,3] — a = [2,3].
a.shift        # => 2 — a = [3].
a.shift        # => 3 — a = [].
a.shift        # => nil — a = [].
```

9.5.2.7. Tablice jako zbiorы

Klasa `Array` implementuje operatory `&`, `|` i - wykonujące operacje przecięcia, sumy i różnicę zbiorów. Dodatkowo dostępna jest w niej metoda `include?` sprawdzająca obecność wybranej wartości w tablicy. Istnieją nawet metody `uniq` i `uniq!` usuwające duplikaty wartości z tablicy (zbiorы nie pozwalają na duplikaty wartości). Tablica nie jest szybką implementacją zbioru (dlatego należy zainteresować się klasą `Set` z biblioteki standardowej), ale może być wygodnym rozwiązaniem do reprezentacji niewielkich zbiorów:

```
[1,3,5] & [1,2,3]      # => [1,3] — przecięcie zbiorów.
[1,1,3,5] & [1,2,3]    # => [1,3] — duplikaty zostały usunięte.
[1,3,5] | [2,4,6]       # => [1,3,5,2,4,6] — scalenie zbiorów.
```

```

[1,3,5,5] | [2,4,6,6]          # => [1,3,5,2,4,6] — duplikaty zostały usunięte.
[1,2,3] - [2,3]                # => [1] — różnica zbiorów.
[1,1,2,2,3,3] - [2, 3]         # => [1] — nie wszystkie duplikaty zostały usunięte.
small = 0..10.to_a              # Zbiór małych liczb.
even = 0..50.map { |x| x*2}    # Zbiór liczb parzystych.
smalleven = small & even       # Przecięcie zbiorów.
smalleven.include?(8)          # => true — sprawdzanie przynależności do zbioru.
[1, 1, nil, nil].uniq          # => [1, nil] — usunięcie duplikatów. Istnieje też metoda uniq!.

```

Zauważ, że operatory `&`, `|` i `-` nie porządkują elementów w zwracanych tablicach. Należy ich używać wyłącznie wtedy, gdy tablica ma reprezentować nieuporządkowany zbiór wartości.

W Ruby 1.9 klasa `Array` definiuje metody kombinatoryczne dla zbiorów do obliczania permutacji, kombinacji i iloczynu kartezjańskiego:

```

a = [1,2,3]
# Iteruje wszystkie możliwe tablice dwuelementowe (kolejność ma znaczenie).
a.permutation(2) { |x| print x } # Drukuję [1,2][1,3][2,1][2,3][3,1][3,2].
# Iteruje wszystkie możliwe podzbiory dwuelementowe (kolejność nie ma znaczenia).
a.combination(2) { |x| print x } # Drukuję [1, 2][1, 3][2, 3].
# Zwraca iloczyn kartezjański dwóch zbiorów.
a.product(['a','b'])           # => [[1,"a"],[1,"b"],[2,"a"],[2,"b"],[3,"a"],[3,"b"]].
[1,2].product([3,4],[5,6])     # => [[1,3,5],[1,3,6],[1,4,5],[1,4,6] itd.].

```

9.5.2.8. Metody tablic asocjacyjnych

Metody `assoc` i `rassoc` pozwalają na traktowanie zwykłych tablic jako tablic asocjacyjnych. W takim przypadku tablica musi być tablicą tablic, np.:

```
[[klucz1, wartość1], [klucz2, wartość2], [klucz3, wartość3], ...]
```

Klasa `Hash` udostępnia metody konwertujące tablice asocjacyjne na tego typu tablice zagnieżdżone. Metoda `assoc` szuka zagnieżdżonej tablicy, której pierwszy element pasuje do podanego argumentu. Zwraca pierwszą pasującą zagnieżdżoną tablicę. Metoda `rassoc` robi to samo, ale zwraca pierwszą zagnieżdżoną tablicę, której drugi element pasuje do argumentu:

```

h = { :a => 1, :b => 2}      # Początkowa tablica asocjacyjna.
a = h.to_a                     # => [[:b,2], [:a,1]] — tablica asocjacyjna.
a.assoc(:a)                   # => [:a,1] — podtablica klucza :a.
a.assoc(:b).last               # => 2 — wartość klucza :b.
a.rassoc(1)                    # => [:a,1] — podtablica wartości 1.
a.rassoc(2).first              # => :b — klucz wartości 2.
a.assoc(:c)                   # => nil.
a.transpose                   # => [[:a, :b], [1, 2]] — zamiana miejscami wierszy i kolumn.

```

9.5.2.9. Różne metody tablicowe

Klasa `Array` udostępnia jeszcze kilka innych metod, które nie pasują do żadnej z wymienionych dotychczas kategorii:

```

# Konwersja nałańcuchy.
[1,2,3].join                  # => 123 — konwersja elementów nałańcuch i połączenie ich.
[1,2,3].join(", ")            # => 1, 2, 3 — opcjonalny znak oddzielający.
[1,2,3].to_s                  # => [1, 2, 3] w Ruby 1.9.
[1,2,3].to_s                  # => 123 w Ruby 1.8.
[1,2,3].inspect                # => [1, 2, 3] — lepsze do debugowania w Ruby 1.8.
# Konwersja binarna za pomocą metody pack. Zobacz także metodę String.unpack.
[1,2,3,4].pack("CCCC")        # => "\001\002\003\004".
[1,2].pack('s2')              # => "001\000\002\000".
[1234].pack("i")              # => "\322\004\000\000".
# Inne metody.
[0,1]*3                      # => [0,1,0,1,0,1] — operator * odpowiada za powtarzanie.

```

```
[1, [2, [3]]].flatten      # => [1,2,3] — rekursywne spłaszczenie. Istnieje też metoda flatten!
[[1, [2, [3]]].flatten(1) # => [[1,2,[3]]] — określa liczbę poziomów. Ruby 1.9.
[1,2,3].reverse          # => [3,2,1] — istnieje też metoda reverse!
a=[1,2,3].zip([:a,:b,:c])# => [[1,.a],[2,.b],[3,.c]] — metoda modułu Enumerable.
a.transpose              # => [[1,2,3],[.a,.b,.c]] — zamienia miejscami wiersze i kolumny.
```

9.5.3. Tablice asocjacyjne

Tablice asocjacyjne zostały wprowadzone w podrozdziale 3.4. Została tam opisana składnia literałów tablic asocjacyjnych (haszowych) oraz operatory `[]` i `[]=` pozwalające pobierać i wstawiać pary klucz-wartość do tablic asocjacyjnych. Niniejszy podrozdział opisuje bardziej szczegółowo API klasy Hash. Tablice asocjacyjne wykorzystują te same operatory złożone z nawiasów kwadratowych co tablice jednowymiarowe i, jak się niebawem przekonasz, wiele metod klasy Hash działa podobnie do metod klasy Array.

9.5.3.1. Tworzenie tablic asocjacyjnych

Tablicę asocjacyjną można utworzyć przy użyciu literala, metody `Hash.new` lub operatora `[]` klasy Hash:

```
{ :one => 1, :two => 2 }    # Podstawowa składnia literalu tablicy asocjacyjnej.
{ :one, 1, :two, 2 }          # To samo przy użyciu wycofywanej składni Ruby 1.8.
{ one: 1, two: 2 }            # To samo przy użyciu składni Ruby 1.9. Klucze są symbolami.
{}                           # Nowy pusty obiekt klasy Hash.
Hash.new                      # => {} — tworzy pustą tablicę asocjacyjną.
Hash[:one, 1, :two, 2]         # => {one:1, two:2}.
```

Przypomnijmy sobie z podrozdziału 6.4.4, że jeśli literał tablicy asocjacyjnej jest ostatnim argumentem wywołania metody, można opuścić jego nawiasy klamrowe:

```
puts :a=>1, :b=>2    # Nawiasy klamrowe opuszczone w wywołaniu.
puts a:1, b:2           # Składnia Ruby 1.9 jest także dozwolona.
```

9.5.3.2. Indeksowanie tablic asocjacyjnych i sprawdzanie przynależności elementów do tablicy

Tablice asocjacyjne bardzo szybko wyszukują wartości związane z kluczami. Możliwe jest też (choć już nie takie szybkie) znalezienie klucza, z którym jest związana określona wartość. Należy jednak pamiętać, że wiele kluczy może odpowiadać takiej samej wartości, w takim przypadku zwracany jest jeden z tych kluczy:

```
h = { :one => 1, :two => 2 }
h[:one]           # => 1 — znajduje wartość skojarzoną z kluczem.
h[:three]         # => nil — nie ma takiego klucza w tablicy.
h.assoc :one     # => [:one, 1] — znajduje parę klucz-wartość. Ruby 1.9.
h.index 1         # => :one — szuka klucza skojarzonego z podaną wartością.
h.index 4         # => nil — nie ma żadnego klucza dla tej wartości.
h.rassoc 2        # => [:two, 2] — para klucz-wartość pasująca do wartości. Ruby 1.9.
```

Klasa Hash udostępnia kilka synonimicznych metod sprawdzających przynależność elementów do tablicy:

```
h = { :a => 1, :b => 2 }
# Sprawdzanie obecności kluczy w tablicy asocjacyjnej — szybkie.
h.key?(:a)           # true — :a jest kluczem w tablicy h.
h.has_key?(:b)        # true — has_key? jest synonimem metody key?.
h.include?(:c)        # false — include? jest kolejnym synonimem.
h.member?(:d)         # false — member? to jeszcze jeden synonim.
# Sprawdzanie obecności wartości — wolne.
h.value?(1)           # true — tablica h zawiera wartość 1.
h.has_value?(3)        # false — has_value? jest synonimem metody value?.
```

Alternatywą dla operatora [] do pobierania wartości z tablicy asocjacyjnej jest metoda `fetch`. Pozwala ona podjąć odpowiednie działania, gdy w tablicy nie ma określonego klucza:

```
h = { :a => 1, :b => 2 }
h.fetch(:a)      #=> 1 — działa jak operator [] dla istniejących kluczów.
h.fetch(:c)      # Zgłasza wyjątek IndexError dla nieistniejącego klucza.
h.fetch(:c, 33)  #=> 33 — używa podanej wartości, jeśli klucz nie zostanie znaleziony.
h.fetch(:c) { |k| k.to_s } #=> c — wywołuje blok, jeśli klucz nie zostanie znaleziony.
```

Aby wydobyć z tablicy asocjacyjnej więcej niż jedną wartość za jednym razem, należy użyć metody `values_at`:

```
h = { :a => 1, :b => 2, :c => 3 }
h.values_at(:c)          #=> [3] — wartości zwrocone w tablicy jednowymiarowej.
h.values_at(:a, :b)       #=> [1, 2] — przekazanie dowolnej liczby argumentów.
h.values_at(:d, :d, :a)   #=> [nil, nil, 1].
```

Klucze i wartości wybrane przez blok można wydobyć za pomocą metody `select`:

```
h = { :a => 1, :b => 2, :c => 3 }
h.select { |k,v| v % 2 == 0 } #=> { :b, 2 } Ruby 1.8.
h.select { |k,v| v % 2 == 0 } #=> { :b => 2 } Ruby 1.9.
```

Niniejsza metoda przesyłania metodę `Enumerable.select`. W Ruby 1.8 metoda `select` zwraca tablicę par klucz-wartość. Natomiast w Ruby 1.9 zwraca tablicę asocjacyjną wybranych kluczów i wartości.

9.5.3.3. Zapisywanie kluczów i wartości w tablicach asocjacyjnych

Do skojarzenia wartości z kluczem w tablicy asocjacyjnej służy operator `[]=` i jego synonim — metoda `store`:

```
h = {}           # Początkowa pusta tablica asocjacyjna.
h[:a] = 1        # Map :a=>1. Teraz h = { :a => 1 }.
h.store(:b, 2)  # Bardziej rozwlekły sposób. Teraz h = { :a => 1, :b => 2 }.
```

Aby zastąpić wszystkie pary klucz-wartość w tablicy asocjacyjnej parami z innej tablicy asocjacyjnej, należy użyć metody `replace`:

```
# Zastępuje wszystkie pary klucz wartość w tablicy h parami z innej tablicy.
h.replace({1=>:a, 2=>:b})  # Teraz tablica h jest równa tablicy podanej jako argument.
```

Metody `merge`, `merge!` i `update` pozwalają na połączenie par z dwóch tablic asocjacyjnych:

```
# Scalenie tablic asocjacyjnych h i j i zapisanie ich w nowej tablicy k.
# Jeśli tablice h i j mają takie same klucze, zostaną dla nich użyte wartości z tablicy j.
k = h.merge(j)
{ :a=>1, :b=>2 }.merge(:a=>3, :c=>3) #=> { :a=>3, :b=>2, :c=>3 }.
h.merge!(j)  # Modyfikuje tablicę h na miejscu.
# Jeśli został podany blok, za jego pomocą nastąpi rozstrzygnięcie, której wartości użyć.
h.merge!(j) { |key,h,j| h }      # Użycie wartości z tablicy h.
h.merge(j) { |key,h,j| (h+j)/2 } # Użycie średniej obu wartości.
# Metoda update jest synonimem metody merge!
h = { a:1,b:2 }      # Użycie składni Ruby 1.9 i opuszczenie klamer.
h.update(b:4,c:9) { |key,old,new| old } # Teraz h = { a:1, b:2, c:9 }.
h.update(b:4,c:9) # Teraz h = { a:1, b:4, c:9 }.
```

9.5.3.4. Usuwanie elementów tablicy asocjacyjnej

Nie można usunąć klucza z tablicy asocjacyjnej, ustawiając go na wartość `nil`. Zamiast tego należy użyć metody `delete`:

```

h = { :a=>1, :b=>2 }
h[:a] = nil           # Teraz tablica h = { :a=>nil, :b=>2 }.
h.include? :a         # => true.
h.delete :b           # => 2 — zwraca usuniętą wartość. Teraz tablica h = { :a=>nil }.
h.include? :b         # => false.
h.delete :b           # => nil — klucz nie został znaleziony.
# Wywołanie bloku, jeśli klucz nie zostanie znaleziony.
h.delete(:b) { |k| raise IndexError, k.to_s } # IndexError!

```

Aby usunąć kilka par klucz-wartość z tablicy asocjacyjnej, należy użyć iteratorów `delete_if` i `reject!` (oraz iteratora `reject` operującego na kopii swojego odbiorcy). Należy zauważać, że metoda `reject` przesyła metodę o tej samej nazwie z modułu `Enumerable` i zwraca tablicę asocjacyjną, a nie tablicę jednowymiarową:

```

h = { :a=>1, :b=>2, :c=>3, :d=>"four" }
h.reject! { |k,v| v.is_a? String }      # => { :a=>1, :b=>2, :c=>3 }.
h.delete_if { |k,v| k.to_s < 'b' }       # => { :b=>2, :c=>3 }.
h.reject! { |k,v| k.to_s < 'b' }       # => nil — brak zmian.
h.delete_if { |k,v| k.to_s < 'b' }       # => { :b=>2, :c=>3 } — tablica nie została zmieniona.
h.reject { |k,v| true }                # => {} — tablica h nie została zmieniona.

```

W końcu, aby usunąć wszystkie pary klucz-wartość z tablicy asocjacyjnej, należy użyć metody `clear`. Metoda ta nie kończy się wykrzyknikiem, ale modyfikuje swojego odbiorcę na miejscu:

```
h.clear      # Teraz tablica h = {}.
```

9.5.3.5. Tablice jednowymiarowe z tablic asocjacyjnych

Klasa `Hash` udostępnia metody konwertujące tablice asocjacyjne na tablice jednowymiarowe:

```

h = { :a=>1, :b=>2, :c=>3 }
# Rozmiar tablicy asocjacyjnej — liczba par klucz-wartość.
h.length      # => 3.
h.size        # => 3 — metoda size jest synonimem metody length.
h.empty?      # => false.
{}.empty?     # => true.
h.keys        # => [ :b, :c, :a ] — tablica jednowymiarowa zawierająca klucze.
h.values      # => [ 2, 3, 1 ] — tablica jednowymiarowa zawierająca wartości.
h.to_a        # => [ [ :b, 2 ], [ :c, 3 ], [ :a, 1 ] ] — tablica jednowymiarowa zawierająca pary.
h.flatten    # => [ :b, 2, :c, 3, :a, 1 ] — spłaszczona tablica jednowymiarowa. Ruby 1.9.
h.sort        # => [ [ :a, 1 ], [ :b, 2 ], [ :c, 3 ] ] — posortowana tablica jednowymiarowa zawierająca pary.
h.sort { |a,b| a[1]<=>b[1] } # Sortowanie par według wartości zamiast kluczy.

```

9.5.3.6. Iteratory tablic asocjacyjnych

Konwertowanie kluczy lub ich wartości nie jest często potrzebne, ponieważ klasa `Hash` umożliwia iterację (dołącza moduł `Enumerable`) i definiuje także inne przydatne iteratory. W Ruby 1.8 nie wiadomo, w jakiej kolejności będą iterowane elementy obiektu klasy `Hash`. Natomiast w Ruby 1.9 elementy iterowane są w kolejności wstawiania. Taka kolejność jest stosowana w powizszych przykładach:

```

h = { :a=>1, :b=>2, :c=>3 }
# Iterator each() iteruje po parach [klucz,wartość].
h.each { |pair| print pair }      # Drukuje [:a, 1][:b, 2][:c, 3].
# Działa także z dwoma argumentami blokowymi.
h.each do |key, value|
  print "#{key}:#{value}"          # Drukuje a:1 b:2 c:3.
end
# Iteracja po kluczach, wartościach lub obu.
h.each_key { |k| print k }        # Drukuje abc.
h.each_value { |v| print v }      # Drukuje 123.
h.each_pair { |k,v| print k, v } # Drukuje a1b2c3. Jak iterator each.

```

Iterator `each` zwraca tablicę jednowymiarową zawierającą klucz i wartość. Składnia wywoływania bloków pozwala na automatyczne rozwinięcie tej tablicy na osobne parametry kluczy i wartości. W Ruby 1.8 iterator `each_pair` zwraca klucz i wartość jako dwie oddzielne wartości (co może nieco przyspieszyć działanie). W Ruby 1.9 metoda `each_pair` jest tylko synonimem metody `each`.

Mimo iż metoda `shift` nie jest iteratorem, można jej używać do iteracji przez pary klucz-wartość tablic asocjacyjnych. Podobnie jak metoda o tej samej nazwie z klasy `Array`, usuwa i zwraca jeden element (jedną parę [klucz,wartość] w tym przypadku) z tablicy asocjacyjnej:

```
h = { :a=> 1, :b=>2 }
print h.shift[1] while not h.empty?      # Drukuję 12.
```

9.5.3.7. Wartości domyślne

Normalnie próba sprawdzenia wartości klucza, z którym nie jest skojarzona żadna wartość, powoduje zwrócenie wartości `nil`:

```
empty = {}
empty["one"]    # nil.
```

Można to jednak zmienić, określając wartość domyślną dla tablicy asocjacyjnej:

```
empty = Hash.new(-1)      # Określenie wartości domyślnej podczas tworzenia tablicy asocjacyjnej.
empty["one"]               # => -1.
empty.default = -2         # Zmiana wartości domyślnej na jakąś inną wartość.
empty["two"]               # => -2.
empty.default             # => -2 — zwrócenie wartości domyślnej.
```

Zamiast podawać pojedynczą wartość domyślną, można dostarczyć blok kodu obliczający wartość dla kluczy, z którymi nie jest skojarzona żadna wartość:

```
# Jeśli ten klucz nie jest zdefiniowany, zwraca następny.
plus1 = Hash.new{|hash, key| keysucc }
plus1[1]          # 2.
plus1["one"]      # "ony" — zobacz metodę String.succ.
plus1.default_proc # Zwraca obiekt klasy Proc obliczający wartości domyślne.
plus1.default(10)   # => 11 — wartość domyślna zwrócona dla klucza 10.
```

Przy zastosowaniu takiego domyślnego bloku często obliczona wartość jest kojarzona z kluczem, dzięki czemu nie trzeba ponownie wykonywać obliczeń, jeśli klucz zostanie użyty jeszcze jeden raz. To jest łatwa w implementacji forma leniwego ewaluowania (wyjaśnia, dlaczego do bloku domyślnego jest przekazywany sam obiekt tablicy asocjacyjnej razem z kluczem):

```
# Niniejsza leniwie inicjowana tablica asocjacyjna rzutuje liczby całkowite na ich silnie:
fact = Hash.new{|h,k| h[k] = if k > 1: k*h[k-1] else 1 end }
fact           # {} — na początku tablica jest pusta.
fact[4]        # 24 — 4! = 24.
fact           # {1=>1, 2=>2, 3=>6, 4=>24} — teraz tablica nie jest pusta.
```

Należy pamiętać, że domyślna własność tablicy asocjacyjnej przesłania blok przekazywany do konstruktora `Hash.new`.

Jeśli nie są potrzebne wartości domyślne lub istnieje potrzeba nadpisania ich własnymi wartościami domyślnymi, do pobierania wartości należy użyć metody `fetch` zamiast kwadratowych nawiasów. Metoda `fetch` została opisana nieco wcześniej:

```
fact.fetch(5)      # IndexError — nie znaleziono klucza.
```

9.5.3.8. Kody mieszające, równość kluczy i klucze modyfikowalne

Aby obiekt można było użyć jako klucza w tablicy asocjacyjnej, musi on udostępniać metodę `hash` zwracającą kod mieszający w postaci liczby całkowitej dla tego obiektu. Klasy niedefiniujące własnej metody `eql?` mogą używać metody `hash` odziedziczonej po klasie `Object`. Jednak definiując metodę `eql?` porównującą obiekty, konieczne jest zdefiniowanie odpowiadającej jej metody `hash`. Jeśli dwa odrębne obiekty są uznawane za równe, ich metody `hash` muszą zwracać tę samą wartość. W idealnej sytuacji dwa obiekty niebędące równymi powinny mieć różne kody mieszające. Temat ten został omówiony w podrozdziale 3.4.2, a podrozdział 7.1.9 zawiera przykładową implementację metody `hash`.

Jak ostrzegaliśmy w podrozdziale 3.4.2, używając modyfikowalnych obiektów jako kluczy w tablicach asocjacyjnych, należy uważać (specjalnym przypadkiem są łańcuchy — klasa `Hash` tworzy prywatne wewnętrzne kopie łańcuchów kluczy). Jeśli jako klucze zostały użyte modyfikowalne obiekty i jeden z nich został zmodyfikowany, należy wywołać metodę `rehash` na rzecz tego obiektu klasy `Hash`, aby zapewnić jego prawidłowe działanie:

```
key = { :a=>1}      # Ta tablica asocjacyjna będzie kluczem w innej tablicy asocjacyjnej!
h = { key => 2 }    # Ta tablica ma modyfikowalny klucz.
h[key]              # => 2 — pobiera wartość związaną z kluczem.
key.clear           # Modyfikacja klucza.
h[key]              # => nil — dla zmodyfikowanego klucza nie znaleziono żadnej wartości.
h.rehash            # Naprawa tablicy asocjacyjnej po modyfikacji klucza.
h[key]              # => 2 — teraz wartość jest znowu znajdowana.
```

9.5.3.9. Różne metody tablic asocjacyjnych

Metoda `invert` nie pasuje do żadnej z wcześniejszych kategorii. Zamienia ona miejscami klucze i wartości w tablicy asocjacyjnej:

```
h = { :a=>1, :b=>2}
h.invert          #=> {1=>:a, 2=>:b} — zamiana miejscami kluczy i wartości.
```

Tak jak w przypadku klasy `Array`, metoda `Hash.to_s` nie jest zbyt przydatna w Ruby 1.8. Dlatego do konwersji na łańcuch w formie literalu tablicy asocjacyjnej lepsza może być metoda `inspect`. W Ruby 1.9 metody `to_s` i `inspect` niczym się nie różnią:

```
{ :a=>1, :b=>2}.to_s      #=> a1b2 w Ruby 1.8; { :a=>1, :b=>2} w 1.9.
{ :a=>1, :b=>2}.inspect #=> { :a=>1, :b=>2} w obu wersjach języka.
```

9.5.4. Zbiory

Zbiór jest kolekcją unikatowych wartości. W przeciwieństwie do tablic elementy zbiorów nie są uporządkowane. Tablicę asocjacyjną można uznać za zbiór par klucz-wartość. Z drugiej strony zbiór można zaimplementować jako tablicę asocjacyjną, w której elementy są kluczem, a wartości są ignorowane. **Zbiór uporządkowany** to taki, w którym elementy są ułożone w jakiejś kolejności (ale nie ma do nich swobodnego dostępu jak w tablicach). Cechą charakterystyczną implementacji zbiorów są szybkie operacje sprawdzania przynależności, wstawiania i usuwania wartości.

W języku Ruby nie ma wbudowanego typu zbioru, ale biblioteka standardowa zawiera klasy `Set` i `SortedSet`, jeśli zostanie dodany wiersz kodu `require 'set'`.

API klasy `Set` jest pod wieloma względami podobne do API klas `Array` i `Hash`. Niektóre metody i operatory klasy `Set` przyjmują jako argument każdy obiekt umożliwiający iterację (`Enumerable`).

Klasa `SortedSet`

Klasa `SortedSet` dziedziczy po klasie `Set` i nie definiuje żadnych własnych metod. Gwarantuje ona tylko, że elementy zbioru będą iterowane (lub drukowane albo konwertowane na tablice jednowymiarowe) w uporządkowanej kolejności. Klasa `SortedSet` nie pozwala na zastosowanie własnego bloku porównującego elementy zbioru i wymaga, aby wszystkie elementy zbioru dały się ze sobą wzajemnie porównywać zgodnie z ich domyślnym operatorem `<=>`. Ponieważ API klasy `SortedSet` niczym nie różni się od API klasy `Set`, nie opisujemy go tutaj.

9.5.4.1. Tworzenie zbiorów

Ponieważ klasa `Set` nie jest rdzenną klasą języka Ruby, nie istnieje składnia literałowa do tworzenia zbiorów. Biblioteka `set` dodaje do modułu `Enumerable` metodę `to_set`, za pomocą której można utworzyć zbiór z każdego obiektu umożliwiającego iterację:

```
(1..5).to_set          # => #<Set: {5, 1, 2, 3, 4}>.  
[1,2,3].to_set        # => #<Set: {1, 2, 3}>.
```

Alternatywnie każdy obiekt umożliwiający iterację można przekazać do metody `Set.new`. Jeśli dostarczony jest blok, zostaje on użyty (jak w przypadku iteratora `map`) do wstępnego przetwarzania wartości przed dodaniem ich do zbioru:

```
Set.new(1..5)          # => #<Set: {5, 1, 2, 3, 4}>.  
Set.new([1,2,3])       # => #<Set: {1, 2, 3}>.  
Set.new([1,2,3]) { |x| x+1 } # => #<Set: {2, 3, 4}>.
```

Aby wyliczyć elementy zbioru bez uprzedniego umieszczenia ich w tablicy lub innym obiekcie umożliwiającym iterację, należy użyć operatora `[]` klasy `Set`:

```
Set["krowa", "świnia", "kura"] # => #<Set: {"krowa", "świnia", "kura"}>.
```

9.5.4.2. Sprawdzanie istnienia wartości, porównywanie i kombinacje zbiorów

Najczęściej wykonywaną operacją na zbiorach jest sprawdzanie, czy zawierają określone wartości:

```
s = Set.new(1..3)    # => #<Set: {1, 2, 3}>.  
s.include? 1         # => true.  
s.member? 0          # => false — metoda member? jest synonimem.
```

Możliwe jest też sprawdzanie przynależności zbiorów do innych zbiorów. Zbiór `S` jest **podzbiorem** zbioru `T`, jeśli wszystkie elementy zbioru `S` znajdują się także w zbiorze `T`. Można także powiedzieć, że `T` jest **nadzbiorem** zbioru `S`. Jeśli dwa zbiory są sobie równe, to są zarówno swoimi nadzbiorami, jak i podzbiorami. `S` jest **podzbiorem właściwym** zbioru `T`, jeśli jest jego podzbiorem, ale nie jest mu równy. W takim przypadku `T` jest **nadzbiorem właściwym** zbioru `S`:

```
s = Set[2, 3, 5]  
t = Set[2, 3, 5, 7]  
s.subset? t          # => true.  
t.subset? s          # => false.
```

```
s.proper_subset? t      # => true.
t.superset? s          # => true.
t.proper_superset? s   # => true.
s.subset? s            # => true.
s.proper_subset? s     # => false.
```

Klasa Set udostępnia taką samą metodę size co klasy Array i Hash:

```
s = Set[2, 3, 5]
s.length               # => 3.
s.size                 # => 3 — synonim metody length.
s.empty?               # => false.
Set.new.empty?         # => true.
```

Nowy zbiór można utworzyć jako kombinację dwóch istniejących już zbiorów. Można to zrobić na kilka sposobów, a służą do tego operatory klasy Set &, |, - oraz ^ (a także ich aliasy w postaci metod):

```
# Dwa proste zbiory.
primes = Set[2, 3, 5, 7]
odds = Set[1, 3, 5, 7, 9]
# Przecięcie to zbiór wartości występujących w obu zbiorach.
primes & odds           # => #<Set: {5, 7, 3}>.
primes.intersection(odds) # To jest alias operatora przecięcia.
# Suma jest zbiorem wartości znajdujących się w każdym ze zbiorów.
primes | odds             # => #<Set: {5, 1, 7, 2, 3, 9}>.
primes.union(odds)        # Alias.
# a-b — elementy zbioru a z wyjątkiem tych, które znajdują się w zbiorze b.
primes-odds                # => #<Set: {2}>.
odds-primes                # => #<Set: {1, 9}>.
primes.difference(odds)    # Alias.
# a^b to zbiór wartości, które znajdują się w jednym ze zbiorów, ale nie w obu — (a|b)-(a&b).
primes ^ odds               # => #<Set: {1, 2, 9}>.
```

Klasa Set udostępnia także modyfikujące wersje niektórych z tych metod. Opisujemy je nieco dalej.

9.5.4.3. Dodawanie i usuwanie elementów

Niniejszy podrozdział opisuje metody dodające elementy do zbiorów lub usuwające elementy ze zbiorów. Są to metody modyfikujące zbiory, na rzecz których są wywoływane, a więc nie tworzą kopii, oryginał pozostawiając bez zmian. Ponieważ metody te nie mają wersji niemodyfikujących, ich nazwy nie kończą się wykryznikiem.

Operator << dodaje jeden element do zbioru:

```
s = Set[]
s << 1                  # Początkowy pusty zbiór.
s.add 2                  # => #<Set: {1}>.
s << 3 << 4 << 5       # => #<Set: {5, 1, 2, 3, 4}> — można tworzyć łańcuchy.
s.add 3                  # => #<Set: {5, 1, 2, 3, 4}> — wartość nie została zmieniona.
s.add? 6                 # => #<Set: {5, 6, 1, 2, 3, 4}>.
s.add? 3                 # => nil — zbiór nie został zmodyfikowany.
```

Aby dodać do zbioru więcej niż jedną wartość, należy użyć metody merge przyjmującej jako argument każdy obiekt umożliwiający iterację. Metoda merge jest w efekcie modyfikującą wersją metody union:

```
s = (1..3).to_set      # => #<Set: {1, 2, 3}>.
s.merge(2..5)           # => #<Set: {5, 1, 2, 3, 4}>.
```

Aby usunąć jeden element ze zbioru, należy użyć metody `delete` lub `delete?`. Metody te są analogiczne do metod `add` i `add?`, ale nie mają ekwiwalentów w postaci operatorów:

```
s = (1..3).to_set    #=> #<Set: {1, 2, 3}>.  
s.delete 1          #=> #<Set: {2, 3}>.  
s.delete 1          #=> #<Set: {2, 3}> — bez zmian.  
s.delete? 1         #=> nil — zwraca wartość nil, kiedy nie ma żadnych zmian.  
s.delete? 2         #=> #<Set: {3}> — w przeciwnym przypadku zwraca zbiór.
```

Do usuwania kilku elementów naraz służy metoda `subtract`. Argumentem tej metody może być każdy obiekt umożliwiający iterację. Działa ona jak modyfikująca wersja metody `difference`:

```
s = (1..3).to_set    #=> #<Set: {1, 2, 3}>.  
s.subtract(2..10)   #=> #<Set: {1}>.
```

Aby usunąć tylko wybrane elementy ze zbioru, należy użyć metod `delete_if` i `reject!`. Podobnie jak w klasach `Array` i `Hash`, metody te są ekwiwalentne z różnicą w wartości zwrotnej, jeśli zbiór nie zostanie zmodyfikowany. Metoda `delete_if` zawsze zwraca zbiór, na rzecz którego została wywołana. Metoda `reject!` zwraca zbiór, na rzecz którego została wywołana, jeśli został on zmodyfikowany, lub wartość `nil`, jeśli nie zostały usunięte z niego żadne wartości:

```
primes = Set[2, 3, 5, 7]      # Zbiór liczb pierwszych.  
primes.delete_if { |x| x%2==1} #=> #<Set: {2}> — usuwa liczby nieparzyste.  
primes.delete_if { |x| x%2==1} #=> #<Set: {2}> — bez zmian.  
primes.reject! { |x| x%2==1}  #=> nil — bez zmian.  
  
# Przecięcie na miejscu:  
s = (1..5).to_set  
t = (4..8).to_set  
s.reject! { |x| not t.include? x} #=> #<Set: {5, 4}>.
```

W końcu metody `clear` i `replace` działają tak samo jak w klasach `Array` i `Hash`:

```
s = Set.new(1..3) # Zbiór początkowy.  
s.replace(3..4)   # Zamiana wszystkich elementów. Argumentem może być każdy obiekt umożliwiający iterację.  
s.clear         #=> #<Set: {}>.  
s.empty?        #=> true.
```

9.5.4.4. Iteratory zbiorów

Zbiory umożliwiają iterację, a klasa `Set` definiuje iterator `each` zwracający jeden raz każdy element zbioru. W Ruby 1.9 klasa `Set` zachowuje się tak samo jak klasa `Hash`, na bazie której została zbudowana, i iteruje po elementach w takiej kolejności, w jakiej zostały wstawione. Przed Ruby 1.9 kolejność iteracji była dowolna. W klasie `SortedSet` elementy są zwracane w kolejności rosnącej. Dodatkowo iterator `map!` transformuje każdy element zbioru za pomocą bloku, modyfikując ten zbiór na miejscu. Synonimem tej metody jest metoda `collect!`:

```
s = Set[1, 2, 3, 4, 5] #=> #<Set: {5, 1, 2, 3, 4}>.  
s.each { |x| print x } # Drukuję 51234 — przed Ruby 1.9 kolejność była dowolna.  
s.map! { |x| x*x }     #=> #<Set: {16, 1, 25, 9, 4}>.  
s.collect! { |x| x/2 } #=> #<Set: {0, 12, 2, 8, 4}>.
```

9.5.4.5. Różne metody zbiorów

Klasa `Set` udostępnia metody dzielące zbiory na podzbiory i spłaszczające zbiory podzbiorów na pojedyncze większe zbiory. Dodatkowo zawiera kilka bardziej przyziemnych metod, którymi zajmiemy się najpierw:

```

s = (1..3).to_set
s.to_a          #=> [1, 2, 3].
s.to_s          #=> "#<Set:0xb7e8f938>" — bezużyteczne.
s.inspect       #=> "#<Set: {1, 2, 3}>" — użyteczne.
s == Set[3, 2, 1] #=> true — porównuje elementy zbioru za pomocą metody eql?.

```

Metoda `classify` wymaga bloku kodu, do którego przesyła kolejno wszystkie elementy zbioru. Wartością zwrotną jest tablica asocjacyjna rzutująca wartości zwrotne bloku na zbiory elementów, które zwróciły te wartości:

```

# Klasifikowanie elementów zbioru jako parzyste lub nieparzyste.
s = (0..3).to_set      #=> #<Set: {0, 1, 2, 3}>.
s.classify { |x| x%2 } #=> {0=>#<Set: {0, 2}>, 1=>#<Set: {1, 3}>}.

```

Metoda `divide` działa podobnie, ale zwraca zbiór podzbiorów zamiast tablicy asocjacyjnej rzutującej wartości na podzbiory:

```
s.divide { |x| x%2 } #=> #<Set: {#<Set: {0, 2}>, #<Set: {1, 3}>}>.
```

Metoda `divide` całkiem zmienia swoje działanie, jeśli związany z nią blok wymaga dwóch argumentów. W takim przypadku blok powinien zwracać wartość `true`, jeśli wartości te należą do tego samego zbioru, lub `false` w przeciwnym przypadku:

```

s = %w[ant ape cow hen hog].to_set #Zbiór słów.
s.divide { |x, y| x[0] == y[0] }        #Dzielenie na podzbiory według pierwszej litery.
#=> #<Set:{#<Set:{"hog", "hen"}>, #<Set:{"cow"}>, #<Set:{"ape", "ant"}>}>.

```

Mając zbiór zbiorów (które same mogą zawierać zbiory), można go spłaszczyć, łącząc (poprzez sumę zbiorów) wszystkie zawarte w nim zbiory za pomocą metody `flatten` lub `flatten!`, która wykonuje operację na miejscu:

```

s = %w[ant ape cow hen hog].to_set #Zbiór słów.
t = s.divide { |x, y| x[0] == y[0] } #Podzielenie zbioru na podzbiory.
t.flatten!                         #Spłaszczenie podzbiorów.
t == s                             #=> true.

```

9.6. Pliki i katalogi

Klasa `File` udostępnia sporą liczbę metod klasowych do pracy na plikach. Dostępne są metody sprawdzające wielkość i istnienie pliku o określonej nazwie czy pozwalające oddzielić nazwę pliku od poprzedzającej ją nazwy katalogu. Są to metody klasowe, a więc nie działają one na obiektach klasy `File`. W zamian nazwy plików podawane są jako łańcuchy. Podobnie klasa `Dir` udostępnia metody klasowe pozwalające na odczytywanie i przetwarzanie plików z katalogów systemu plików. Kolejne podrozdziały opisują:

- przetwarzanie i manipulację nazwami plików i katalogów,
- tworzenie list zawartości katalogów,
- sprawdzanie typu, rozmiaru, czasu ostatniej modyfikacji i innych atrybutów plików,
- usuwanie, zmianianie nazwy i wykonywanie innych tego typu operacji na plikach i katalogach.

Zwróc uwagę, że opisane tutaj metody wysyłają zapytania do plików i manipuluują nimi, ale nie odczytują ani nie zapisują ich **treści**. Odczyt i zapis plików został opisany w podrozdziale 9.7.

Określanie nazw plików w Ruby 1.9

Wiele z opisanych w niniejszym podrozdziale metod przyjmuje jeden lub więcej argumentów wyznaczających nazwy plików. Normalnie nazwy plików i ścieżki katalogów określane są jako łańcuchy. W Ruby 1.9 można używać także obiektów niebędących łańcuchami, jeśli udostępniają one metodę `to_path` zwracającą łańcuch.

9.6.1. Nazwy plików i katalogów

Metody klasowe klas `File` i `Dir` działają na plikach i katalogach wyznaczonych za pomocą nazw. W Ruby stosowane są nazwy plików w stylu systemu Unix ze znakiem / jako separatorem katalogów. Ukośnika tego można używać nawet w systemie Windows. W systemie tym Ruby obsługuje też ścieżki do plików zawierające lewy ukośnik i prefiks w postaci litery dysku. Stała `File::SEPARATOR` powinna mieć wartość / we wszystkich implementacjach. Stała `File::ALT_SEPARATOR` ma wartość \ w systemie Windows i nil na innych platformach.

Klasa `File` udostępnia kilka metod do manipulowania nazwami plików:

```
full = '/home/matz/bin/ruby.exe'  
file=File.basename(full)      # => ruby.exe — tylko nazwa lokalna.  
File.basename(full, '.exe')   # => ruby — usunięte rozszerzenie.  
dir=File.dirname(full)        # => /home/matz/bin — bez znaku / na końcu.  
File.dirname(file)           # => . — aktualny katalog.  
File.split(full)             # => ['/home/matz/bin', 'ruby.exe'].  
File.extname(full)           # => .exe.  
File.extname(file)           # => .exe.  
File.extname(dir)            # => "  
File.join('home', 'matz')    # => home/matz — względna.  
File.join('', 'home', 'matz') # => /home/matz bezwzględna.
```

Metoda `File.expand_path` konwertuje ścieżkę względną na pełną ścieżkę. Jeśli zostanie podany opcjonalny drugi argument, zostanie on dołączony jako nazwa katalogu z przodu pierwszego argumentu. Następnie cały wynik zostanie przekonwertowany na ścieżkę bezwzględną. Jeśli zaczyna się od znaku ~ (w stylu systemu Unix), katalog jest położony względem katalogu głównego aktualnego lub określonego użytkownika. W przeciwnym przypadku katalog jest odnajdywany względem aktualnego katalogu roboczego (zobacz opisaną poniżej metodę `Dir.chdir` zmieniającą katalog roboczy):

```
Dir.chdir("/usr/bin")          # Aktualny katalog roboczy to /usr/bin.  
File.expand_path("ruby")        # => /usr/bin/ruby.  
File.expand_path("~/ruby")      # => /home/david/ruby.  
File.expand_path("~/matz/ruby") # => /home/matz/ruby.  
File.expand_path("ruby", "/usr/local/bin") # => /usr/local/bin/ruby.  
File.expand_path("ruby", "../local/bin")   # => /usr/local/bin/ruby.  
File.expand_path("ruby", "~/.bin")        # => /home/david/bin/ruby.
```

Metoda `File.identical?` sprawdza, czy dwie nazwy pliku odnoszą się do tego samego pliku. Może się to zdarzyć, gdy nazwy są takie same, ale metoda ta przydaje się bardziej wówczas, gdy nazwy się różnią, na przykład jedna jest względna, a druga bezwzględna. Ktoś mógł dodać znaki .., aby przejść do góry o jeden poziom, a następnie do dołu. Dwie różne nazwy mogą odnosić się do tego samego pliku, jeśli jedna z nich jest dowiązaniem symbolicznym lub skrótem (lub dowiązaniem twardym na platformach je obsługujących) do drugiej. Należy jednak pamiętać, że metoda `File.identical?` zwraca wartość true tylko wtedy, gdy obie nazwy odnoszą się do tego samego pliku i plik ten istnieje. Ponadto należy zauważyc, że metoda `File.identical?` nie rozwiązuje znaku ~ w taki sam sposób jak metoda `File.expand_path`:

```

File.identical?("ruby", "ruby")           # => true, jeśli plik ten istnieje.
File.identical?("ruby", "/usr/bin/ruby")   # => true, jeśli bieżący katalog roboczy to /usr/bin.
File.identical?("ruby", "../bin/ruby")     # => true, jeśli bieżący katalog roboczy to /usr/bin.
File.identical?("ruby", "ruby1.9")         # => true, jeśli istnieje dowiezanie.

```

W końcu metoda `File.fnmatch` sprawdza, czy nazwa pliku pasuje do określonego wzorca. Wzorzec ten nie jest wyrażeniem regularnym, a przypomina raczej wzorce dopasowujące nazwy plików używane w powłokach. Znak `?` dopasowuje jeden znak, `*` dopasowuje dowolną liczbę znaków, `**` dopasowuje dowolną liczbę poziomów katalogów. Znaki znajdujące się w nawiasach kwadratowych oznaczają alternatywy, podobnie jak w wyrażeniach regularnych. Metoda `fnmatch` nie pozwala na wstawianie alternatyw do nawiasów klamrowych (w przeciwieństwie do metody `Dir.glob`, która została opisana dalej). Metoda `fnmatch` powinna być wywoływana zazwyczaj z trzecim argumentem `File::FNM_PATHNAME`, który zapobiega dopasowaniu przez `*` znaku `/`. Aby dopasowywane były także ukryte pliki, których nazwy zaczynają się od znaku `.`, należy dodać stałą `File::FNM_DOTMATCH`. Przedstawiamy tylko kilka przykładów użycia metody `fnmatch`. Więcej na jej temat można znaleźć za pomocą polecenia `ri File.fnmatch`. Synonimem tej metody jest `File.fnmatch?`:

```

File.fnmatch("*.rb", "hello.rb")      # => true.
File.fnmatch("*. [ch]", "ruby.c")      # => true.
File.fnmatch("*. [ch]", "ruby.h")      # => true.
File.fnmatch("?.txt", "ab.txt")        # => false.
flags = File::FNM_PATHNAME | File::FNM_DOTMATCH
File.fnmatch("lib/*.rb", "lib/a.rb", flags)    # => true.
File.fnmatch("lib/*.rb", "lib/a/b.rb", flags)   # => false.
File.fnmatch("lib/**/*.*.rb", "lib/a.rb", flags) # => true.
File.fnmatch("lib/**/*.*.rb", "lib/a/b.rb", flags) # => true.

```

9.6.2. Tworzenie listy zawartości katalogów

Najprostszym sposobem na utworzenie listy zawartości katalogu jest użycie metody `Dir.entries` lub iteratora `Dir.foreach`:

```

# tworzy listę wszystkich plików katalogu config/
filenames = Dir.entries("config")          # Pobranie nazw w tablicy.
Dir.foreach("config") { |filename| ... }    # Iteracja nazw.

```

Nie ma żadnej gwarancji, że metody te zwrócią nazwy w jakiejś określonej kolejności i że (w systemach uniksowych) dołączą `.` (bieżący katalog) oraz `..` (katalog nadrzędny). Aby uzyskać listę plików, których nazwy pasują do określonego wzorca, należy użyć operatora `Dir[]`:

```

Dir['*.data']           # Pliki z rozszerzeniem data.
Dir['ruby.*']            # Każdy plik, którego nazwa zaczyna się od słowa ruby.
Dir['?']                 # Każda nazwa pliku złożona z jednego znaku.
Dir['*.[ch]']             # Każdy plik, którego nazwa kończy się znakami .c lub .h.
Dir['*.{java,rb}']        # Każdy plik, którego nazwa kończy się znakami java lub rb.
Dir['*/*.rb']             # Każdy program Ruby w każdym bezpośrednim podkatalogu.
Dir['**/*.rb']            # Każdy program Ruby w jakimkolwiek podkatalogu.

```

Alternatywą o większych możliwościach dla operatora `Dir[]` jest metoda `Dir.glob` (słowo `glob` jest starym terminem z systemu Unix oznaczającym dopasowywanie nazwy pliku w powłoce). Domyślnie metoda ta działa tak samo jak operator `Dir[]`, ale jeśli zostanie podany blok, zwraca dopasowane nazwy plików pojedynczo, zamiast zwracać tablicę. Dodatkowo metoda `glob` przyjmuje opcjonalny drugi argument. Jeśli w tym miejscu zostanie przekazana stała `File::FNM_DOTMATCH` (zobacz opisaną wcześniej metodę `File.fnmatch`), zostaną wzięte pod uwagę też pliki, których nazwy zaczynają się od znaku `.` (w systemie Unix pliki te są ukryte i domyślnie nie są zwracane):

```

Dir.glob('*.rb') {|f| ... }           # Iteracja po wszystkich plikach z kodem Ruby.
Dir.glob('*')                         # Nie bierze pod uwagę plików, których nazwy zaczynają się od kropki.
Dir.glob('*',File::FNM_DOTMATCH)      # Dodaje pliki, których nazwy zaczynają się od kropki, tak jak metoda
Dir.entries.

```

Zaprezentowane tutaj metody zwracające zawartość katalogów oraz wszystkie metody klas `File` i `Dir` rozwijające względne ścieżki do plików działają względem aktualnego katalogu roboczego, który jest wartością globalną do procesu interpretera Ruby. Bieżący katalog roboczy można sprawdzić i ustawić za pomocą metod `getwd` i `chdir`:

```

puts Dir.getwd          # Drukuję aktualny katalog roboczy.
Dir.chdir(..)            # Zmienia katalog roboczy na katalog nadzędny.
Dir.chdir('../sibling')  # Ponownie zmienia na katalog równoległy.
Dir.chdir('/home')        # Zmienia na katalog główny.
Dir.chdir                # Zmienia na katalog główny użytkownika.
home = Dir.pwd           # Metoda pwd jest aliasem metody getwd.

```

Jeśli do metody `chdir` zostanie przekazany blok, katalog ten zostanie przywrócony do swojej pierwotnej wartości po zakończeniu działania bloku. Należy jednak pamiętać, że mimo iż zmiana katalogu jest tylko tymczasowa, ma ona nadal zasięg globalny i wpływa na inne wątki. Dwa wątki nie mogą wywołać metody `Dir.chdir` z blokiem w tym samym czasie.

9.6.3. Sprawdzanie informacji o plikach

Klasa `File` udostępnia mnóstwo metod do sprawdzania metadanych plików i katalogów. Wiele z nich zwraca informacje niskopoziomowe, które zależą od konkretnego systemu operacyjnego. Tutaj przedstawiamy tylko te najbardziej przenośne i przydatne z nich. Pełną listę tych metod można uzyskać, używając narzędzia `ri` na klasach `File` i `File::Stat`.

Poniższe proste metody zwracają podstawowe informacje o pliku. Większość z nich to predykaty zwracające wartości `true` i `false`:

```

f = "/usr/bin/ruby"          # Nazwa pliku używana w poniższych przykładach.
# Imię pliku i typy.
File.exist?(f)               # Czy plik o takiej nazwie istnieje? Także File.exists?.
File.file?(f)                 # Czy to jest istniejący plik?
File.directory?(f)            # Czy jest to istniejący katalog?
File.symlink?(f)              # Czy jest to mimo wszystko dowiązanie symboliczne?
# Rozmiar pliku. Do ustawiania rozmiaru pliku służy metoda Filetruncate.
File.size(f)                  # Rozmiar pliku w bajtach.
File.size?(f)                 # Rozmiar w bajtach lub nil, jeśli plik jest pusty.
File.zero?(f)                 # Prawda, jeśli plik jest pusty.
# Zezwolenia dostępu do plików. Do ustawiania zezwoleń dostępu służy metoda Filechmod (zależna od systemu).
File.readable?(f)              # Czy można odczytywać ten plik?
File.writable?(f)              # Czy można zapisywać w tym pliku?
File.executable?(f)            # Czy można ten plik wykonywać?
File.world_readable?(f)         # Czy każdy może odczytać ten plik? Ruby 1.9.
File.world_writable?(f)         # Czy każdy może zapisywać w tym pliku? Ruby 1.9.
# Czas. Do ustawiania czasu służy metoda File.ctime.
File.mtime(f)                  #=> Czas ostatniej modyfikacji w postaci obiektu klasy Time.
File.atime(f)                  #=> Czas ostatniego dostępu do pliku w postaci obiektu klasy Time.

```

Innym sposobem na określenie typu (plik, katalog, dowiązanie symboliczne itd.) nazwy pliku jest użycie metody `ftype` zwracającej łańcuch wyznaczający typ. Założmy, że `/usr/bin/ruby` jest dowiązaniem symbolicznym (lub skrótem) do `/usr/bin/ruby1.9`:

```

File.ftype("/usr/bin/ruby")    #=> link.
File.ftype("/usr/bin/ruby1.9")  #=> file.
File.ftype("/usr/lib/ruby")     #=> directory.
File.ftype("/usr/bin/ruby3.0")   # SystemCallError — nie ma takiego pliku lub katalogu.

```

Jeśli potrzebnych jest dużo informacji o pliku, lepiej jest użyć metody `stat` lub `lstat` (metoda `stat` podąża za dowiązaniem symbolicznym, a `lstat` zwraca informacje o samych dowiązaniach). Metody te zwracają obiekt klasy `File::Stat` udostępniający metody egzemplarza o takich samych nazwach (ale bez argumentów) jak nazwy metod klasy `File`. Zaletą metody `stat` jest to, że potrzebne jest tylko jedno zapytanie do systemu operacyjnego, aby uzyskać wszystkie metadane pliku. Następnie dane te można pobrać w programie z obiektu klasy `File::Stat` w razie potrzeby:

```
s = File.stat("/usr/bin/ruby")
s.file?                      # => true.
s.directory?                 # => false.
s.ftype                       # => "file".
s.readable?                   # => true.
s.writable?                   # => false.
s.executable?                 # => true.
s.size                        # => 5492.
s.atime                       # => Mon Jul 23 13:20:37 -0700 2007.
```

Pełną listę metod klasy `File::Stat` można uzyskać za pomocą narzędzia `ri`. Ostatnią ogólną metodą do testowania plików, którą opiszemy, jest metoda `Kernel.test`. Istnieje ona ze względu na historyczną zgodność z poleceniem `test` powłoki uniksowej. Metoda ta jest w znacznym stopniu wypierana przez metody klasy `File`, ale czasami spotyka się ją w niektórych skryptach Ruby. Szczegółowych informacji na jej temat dostarcza narzędzie `ri`:

```
# Sprawdzanie pojedynczych plików.
test ?e, "/usr/bin/ruby"      # File.exist?("usr/bin/ruby").
test ?f, "/usr/bin/ruby"      # File.file?("usr/bin/ruby").
test ?d, "/usr/bin/ruby"      # File.directory?("usr/bin/ruby").
test ?r, "/usr/bin/ruby"      # File.readable?("usr/bin/ruby").
test ?w, "/usr/bin/ruby"      # File.writeable?("usr/bin/ruby").
test ?M, "/usr/bin/ruby"      # File.mtime("usr/bin/ruby").
test ?s, "/usr/bin/ruby"      # File.size?("usr/bin/ruby").
# Porównywanie dwóch plików f,g.
test ?-, f, g                # File.identical(f,g).
test ?<, f, g                # File(f).mtime < File(g).mtime.
test ?>, f, g                # File(f).mtime > File(g).mtime.
test ?=, f, g                # File(f).mtime == File(g).mtime.
```

9.6.4. Tworzenie, usuwanie

oraz zmienienie nazw plików i katalogów

Klasa `File` nie definiuje żadnych specjalnych metod do tworzenia plików. Aby utworzyć plik, należy go otworzyć do zapisu, zapisać zero lub więcej bajtów i zamknąć go. Aby nie uszkodzić danych w istniejącym pliku, należy go otworzyć w trybie dodawania:

```
# Utworzenie (lub nadpisanie) pliku o nazwie test.
File.open("test", "w") {}
# Utworzenie (ale nienadpisane) pliku o nazwie test.
File.open("test", "a") {}
```

Aby skopiować plik, należy użyć metody `File.copy_stream`, podając nazwy plików jako argumenty.

```
File.copy_stream("test", "test.backup")
```

Aby zmienić nazwę pliku, należy użyć metody `File.rename`:

```
File.rename("test", "test.old")      # Aktualna nazwa i nowa nazwa.
```

Aby utworzyć dowiązanie symboliczne do pliku, należy użyć metody `File.symlink`:

```
File.symlink("test.old", "oldtest") # Cel dowiązania, nazwa dowiązania.
```

W niektórych systemach można utworzyć twarde dowiązanie za pomocą metody `File.link`:

```
File.link("test.old", "test2") # Cel dowiązania, nazwa dowiązania.
```

Aby usunąć plik lub dowiązanie, należy użyć metody `File.delete` lub jej synonimu `File.unlink`:

```
File.delete("test2") # Może być też wywoływana z wieloma argumentami,
```

```
File.unlink("oldtest") # aby usunąć wiele plików.
```

Niektóre systemy umożliwiają przycinanie plików do określonej liczby bajtów. Służy do tego metoda `File.truncate`. Do ustawiania czasu modyfikacji i dostępu do pliku służy metoda `File.utime`. Zezwolenia dostępu do pliku ustawiane są za pomocą zależnej od platformy metody `File.chmod`:

```
f = "log.messages" # Nazwa pliku.  
atime = mtime = Time.now # Nowe czasy dostępu i modyfikacji.  
Filetruncate(f, 0) # Wyczyszczenie całej zawartości.  
File.utime(atime, mtime, f) # Zmiana czasów.  
File.chmod(0600, f) # Zezwolenia w systemie Unix -rw-----. Zwróć uwagę na argument ósemkowy.
```

Aby utworzyć nowy katalog, należy użyć metody `Dir.mkdir`. Aby usunąć katalog, należy użyć metody `Dir.rmdir` lub jednego z jej synonimów — `Dir.delete` lub `Dir.unlink`. Katalog do usunięcia musi być pusty:

```
Dir.mkdir("temp") # Utworzenie katalogu.  
File.open("temp/f", "w") {} # Utworzenie pliku w katalogu.  
File.open("temp/g", "w") {} # Utworzenie jeszcze jednego pliku.  
File.delete(*Dir["temp/*"]) # Usunięcie wszystkich plików z katalogu.  
Dir.rmdir("temp") # Usunięcie katalogu.
```

9.7. Wejście i wyjście

Obiekt klasy `IO` jest strumieniem — dającym się odczytać źródłem bajtów lub znaków albo umożliwiającym zapis zbiornikiem na bajty lub znaki. Klasa `File` jest podklassą klasy `IO`. Obiekty klasy `IO` reprezentują także standardowy strumień wejściowy i standardowy strumień wyjściowy używane do odczytu danych z konsoli i wysyłania danych do konsoli. Moduł `stringio` z biblioteki standardowej pozwala na tworzenie strumieniowych osłon na obiektach łańcuchowych. W końcu obiekty gniazd używane w programowaniu sieciowym (opisane dalej) również są obiektami klasy `IO`.

9.7.1. Otwieranie strumieni

Przed pobraniem danych wejściowych lub wysłaniem danych na wyjście konieczne jest utworzenie obiektu klasy `IO`, z którego będzie następował odczyt lub do którego będą zapisywane dane. Klasa `IO` udostępnia metody fabrykujące `new`, `open`, `popen` i `pipe`. Są to jednak metody niskiego poziomu, zależne od systemu operacyjnego, i nie będziemy ich tutaj opisywać. Kolejne podrozdziały opisują częściej stosowane sposoby tworzenia obiektów klasy `IO` (a podrozdział 9.8 zawiera przykłady tworzenia obiektów klasy `IO` komunikujących się przez sieć).

9.7.1.1. Otwieranie plików

Jednym z najczęściej używanych rodzajów operacji wejścia i wyjścia jest odczyt i zapis plików. Klasa `File` definiuje kilka metod użytkowych (opisane poniżej), które odczytują całą zawartość pliku w jednym wywołaniu. Często jednak zamiast tego otwiera się plik w celu uzyskania obiektu klasy `File` i następnie za pomocą metody klasy `IO` odczytuje się jego zawartość lub zapisuje w nim dane.

Aby otworzyć plik, należy użyć metody `File.open` (lub `File.new`). Pierwszy argument jest nazwą pliku. Nazwa ta jest zazwyczaj podawana w postaci łańcucha, ale w Ruby 1.9 można użyć dowolnego obiektu udostępniającego metodę `to_path`. Nazwy plików są interpretowane względem bieżącego katalogu roboczego, chyba że zostaną określone jako ścieżki bezwzględne. Do oddzielania nazw katalogów służy ukośnik — w systemie Windows są one automatycznie konwertowane na lewe ukośniki. Drugim argumentem metody `File.open` jest krótki łańcuch określający sposób otwarcia pliku:

```
f = File.open("data.txt", "r")    # Otwarcie pliku data.txt do odczytu.  
out = File.open("out.txt", "w")   # Otwarcie pliku out.txt do zapisu.
```

Drugi argument metody `File.open` jest łańcuchem określającym „tryb pliku”. Musi się on zaczynać od jednej z wartości wymienionych w poniższej tabeli. Dodanie opcji `b` do łańcucha trybu zapobiega automatycznej konwersji znaku końca wiersza w systemie Windows. W przypadku plików tekstowych do łańcucha trybu można dodać nazwę kodowania znaków. Pliki binarne należy otwierać w trybie `:binary`. Więcej informacji na ten temat znajduje się w podrozdziale 9.7.2.

Tryb	Opis
r	Otwiera plik do odczytu. Tryb domyślny.
r+	Otwiera plik do odczytu i zapisu. Zaczyna od początku pliku. Kończy się niepowodzeniem, jeśli plik nie istnieje.
w	Otwiera plik do zapisu. Tworzy nowy plik lub przycina istniejący.
w+	To samo co w, ale pozwala też na odczyt pliku.
a	Otwiera plik do zapisu, ale jeśli dany plik istnieje, dodaje dane na jego końcu.
a+	To samo co a, ale pozwala też na odczyt.

Po metodzie `File.open` (ale nie po metodzie `File.new`) może znajdować się blok. W takim przypadku metoda `File.open` nie zwraca obiektu klasy `File`, tylko przekazuje go do tego bloku i automatycznie go zamyka w chwili zakończenia działania bloku. Wartość zwrotna bloku staje się wartością zwrotną metody `File.open`:

```
File.open("log.txt", "a") do |log|          # Otwarcie w trybie dodawania.  
  log.puts("INFO: Zapisywanie komunikatu w dzienniku")  # Wysłanie danych do pliku.  
end
```

9.7.1.2. Metoda Kernel.open

Metoda `open` z modułu `Kernel` działa jak metoda `File.open`, ale jest bardziej elastyczna. Jeśli nazwa pliku zaczyna się od znaku `|`, jest traktowana jako polecenie systemu operacyjnego i zwrócony strumień jest używany do odczytu z i zapisu do procesu tego polecenia. Jest to operacja zależna od platformy:

```
# Ile czasu działa serwer?  
uptime = open("|uptime") {|f| f.gets }
```

Jeśli została załadowana biblioteka `open-uri`, metoda `open` może też czytać z adresów HTTP i FTP, jakby były plikami:

```
require "open-uri"          # Wymagana biblioteka.  
f = open("http://www.davidflanagan.com/") # Strona internetowa jako plik.  
webpage = f.read             # Odczyt jako jednego długiegołańcucha.  
f.close                      # Nie zapomnij o zamknięciu pliku!
```

W Ruby 1.9, jeśli argument przekazany do metody `open` udostępnia metodę `to_open`, wywoływana jest ta metoda i powinna ona zwrócić otwarty obiekt klasy `IO`.

9.7.1.3. Biblioteka `stringio`

Innym sposobem na utworzenie obiektu klasy `IO` jest użycie biblioteki `stringio` do odczytu z łańcucha lub zapisu w łańcuchu:

```
require "stringio"  
input = StringIO.open("now is the time") # Odczyt z tego łańcucha.  
buffer = ""  
output = StringIO.open(buffer, "w")      # Zapis w buforze.
```

Klasa `StringIO` nie jest podklasą klasy `IO`, ale definiuje wiele takich samych metod jak klasa `IO`. Dzięki kaczemu typowaniu często w miejsce obiektów klasy `IO` można używać obiektów klasy `StringIO`.

9.7.1.4. Strumienie predefiniowane

Ruby predefiniuje kilka strumieni, których można używać bez ich tworzenia lub otwierania. Stałe globalne `STDIN`, `STDOUT` i `STDERR` to odpowiednio: standardowy strumień wejścia, standardowy strumień wyjścia i standardowy strumień błędów. Domyślnie strumienie te są połączone z konsolą użytkownika lub jakiegoś rodzaju oknem terminala. W zależności od sposobu uruchomienia skryptu Ruby strumienie te mogą jako źródła danych wejściowych lub miejsca docelowego dla danych wyjściowych używać pliku, a nawet innego procesu. Każdy program Ruby może wczytywać dane ze standardowego strumienia wejściowego i wysyłać dane do standardowego strumienia wyjściowego (w przypadku normalnych danych generowanych przez program) lub standardowego strumienia błędów (w przypadku błędów, które powinny być widoczne nawet, jeśli standardowy strumień wyjściowy jest skierowany do pliku). Zmienne globalne `$stdin`, `$stdout` i `$stderr` są początkowo ustawiane na te same wartości co stałe strumieniowe. Funkcje globalne, takie jak `print` i `puts`, domyślnie zapisują dane w zmiennej `$stdout`. Jeśli skrypt zmieni wartość tej zmiennej globalnej, zmieni się zachowanie tych metod. Jednak „prawdziwy” standardowy strumień wyjściowy będzie nadal dostępny przez stałą `STDOUT`.

Innym predefiniowanym strumieniem jest `ARGF` lub `$<`. Strumień ten ma za zadanie ułatwiać pisanie skryptów odczytujących pliki wyznaczone w wierszu poleceń lub pochodzące ze standardowego strumienia wejściowego. Jeśli istnieją jakieś argumenty wiersza poleceń dla skryptu (w tablicy `ARGV` lub `$*`), strumień `ARGF` działa tak, jakby te strumienie zostały połączone i został otwarty jeden plik do odczytu. Aby nie było żadnych problemów, skrypt Ruby przyjmujący opcje z wiersza poleceń inne niż nazwy plików musi najpierw przetworzyć te opcje i usunąć je z tablicy `ARGV`. Jeśli tablica `ARGV` jest pusta, strumień `ARGF` jest równoważny ze strumieniem `STDIN` (więcej szczegółów na temat strumienia `ARGF` znajduje się w podrozdziale 10.3.1).

W końcu strumień DATA służy do odczytywania tekstu znajdującego się za skryptem Ruby. Działa to tylko, jeśli skrypt zawiera wiersz, w którym nie znajduje się nic poza tokenem _END_. Token ten oznacza koniec kodu źródłowego programu. Wszystko, co znajduje się za tym tokenem, można odczytać za pomocą strumienia DATA.

9.7.2. Strumienie i kodowanie

Jedną z najważniejszych zmian w Ruby 1.9 jest dodanie obsługi kodowań znaków wielobajtowych. W podrozdziale 3.2 przekonałeś się, że w klasie `String` nastąpiły znaczne zmiany. Podobne zmiany nastąpiły w klasie `IO`.

W Ruby 1.9 z każdym strumieniem mogą być związane dwa kodowania. Są one nazywane kodowaniem wewnętrznym i kodowaniem zewnętrznym, a zwracając je metody `internal_encoding` i `external_encoding` z klasy `IO`. Kodowanieowe to kodowanie tekstu zastosowane w pliku. Kodowanie wewnętrzne to kodowanie użyte do reprezentowania tekstu w języku Ruby. Jeśli kodowanie wewnętrzne jest takie samo jak wymagane kodowanie wewnętrzne, nie ma potrzeby określania kodowania wewnętrznego — łańcuchy wczytane ze strumienia będą miały kodowanie wewnętrzne (tak jakby wymuszone przez metodę `force_encoding` z klasy `String`). Jeśli jednak wewnętrzna reprezentacja musi być inna niż wewnętrzna, można określić kodowanie wewnętrzne i Ruby będzie transkodował tekst z kodowania zewnętrznego na wewnętrzne podczas odczytu danych i na wewnętrzne podczas zapisu.

Do określania kodowania obiektów klasy `IO` (wliczając potoki i gniazda sieciowe) służy metoda `set_encoding`. Jeśli zostaną podane do niej dwa argumenty, określa ona kodowanie wewnętrzne i wewnętrzne. Można także określić dwa kodowania za pomocą jednego argumentu łańcuchowego składającego się z dwóch nazw kodowań oddzielonych dwukropkiem. Normalnie jednak pojedynczy argument zazwyczaj określa kodowanie wewnętrzne. Argumentami mogą być łańcuchy lub obiekty klasy `Encoding`. Najpierw zawsze określone jest kodowanie wewnętrzne, po którym opcjonalnie może zostać określone kodowanie wewnętrzne. Na przykład:

```
f.set_encoding("iso-8859-1", "utf-8") # Latin-1, transkodowane na UTF-8.  
f.set_encoding("iso-8859-1:utf-8")      # To samo co powyżej.  
f.set_encoding(Encoding::UTF-8)        # Tekst UTF-8.
```

Metoda `set_encoding` działa z każdym rodzajem obiektów klasy `IO`. Jeśli jednak chodzi o pliki, kodowanie najczęściej najłatwiej jest określić przy otwieraniu pliku. Można to zrobić poprzez dołączenie nazw kodowań do łańcucha trybu pliku. Na przykład:

```
in = File.open("data.txt", "r:utf-8");           # Odczyt tekstu UTF-8.  
out = File.open("log", "a:utf-8");                # Zapis tekstu UTF-8.  
in = File.open("data.txt", "r:iso8859-1:utf-8"); # Konwersja Latin-1 na UTF-8.
```

Należy zauważyć, że zazwyczaj nie ma konieczności określania dwóch kodowań dla strumienia, który ma być używany jako wyjście. W takim przypadku kodowanie wewnętrzne jest określone przez obiekty klasy `String`, które są zapisywane do tego strumienia.

Jeśli nie zostanie określone żadne kodowanie, przy odczytcie plików zostanie zastosowane domyślne kodowanie wewnętrzne języka Ruby (zobacz podrozdział 2.4.2), natomiast przy zapisie do plików oraz odczytcie i zapisie do potoków i gniazd nie zostanie zastosowane żadne kodowanie (tzn. kodowanie ASCII-8BIT/BINARY).

Domyślne kodowanie zewnętrzne jest zgodne z ustawieniami lokalnymi użytkownika i często jest kodowaniem wielobajtowym. Aby odczytać dane binarne z pliku, należy zatem jawnie określić, że bajty mają być niezakodowane, albo otrzymamy znaki w domyślnym kodowaniu zewnętrzny. W tym celu należy otworzyć plik w trybie `r:binary` lub przekazać stałą `Encoding::BINARY` do metody `set_encoding` po otwarciu pliku:

```
File.open("data", "r:binary") # Otwarcie pliku do odczytu danych binarnych.
```

W systemie Windows pliki binarne należy otwierać w trybie `rb:binary` lub na rzecz strumienia wywołać metodę `binmode`. To wyłącza automatyczną konwersję znaków nowego wiersza wykonywaną w systemie Windows i jest wymagane wyłącznie w systemie Windows.

Nie każda metoda czytająca ze strumienia akceptuje jego kodowanie. Niektóre metody niższego poziomu pobierają argument określający liczbę bajtów, które mają zostać odczytane. Metody te z natury zwracają niezakodowane łańcuchy bajtów zamiast łańcuchów tekstowych. Metody nieokreślające liczby bajtów do odczytania honorują kodowanie.

9.7.3. Odczyt ze strumienia

Klasa `IO` udostępnia kilka metod służących do odczytu danych ze strumienia. Oczywiście działają one tylko, gdy strumień jest przeznaczony do odczytu. Można doczytywać dane ze strumieni `STDIN`, `ARGF` i `DATA`, ale nie z `STDOUT` czy `STDERR`. Pliki i obiekty klasy `StringIO` są domyślnie otwierane do odczytu, chyba że jawnie zostaną otwarte w trybie tylko do zapisu.

9.7.3.1. Odczytywanie wierszy danych

Klasa `IO` umożliwia odczyt wierszy danych ze strumienia na kilka sposobów:

```
lines = ARGF.readlines          # Wczytanie wszystkich danych wejściowych i zwrócenie tablicy poszczególnych wierszy.  
line = DATA.readline           # Wczytanie jednego wiersza danych ze strumienia.  
print l while l = DATA.gets    # Odczytywanie, dopóki metoda gets nie zwróci wartości nil po dotarciu na koniec pliku.  
DATA.each {|line| print line } # Iteracja wierszy ze strumienia do napotkania końca pliku.  
DATA.each_line                 # Alias metody each.  
DATA.lines                      # Enumerator dla metody each_line — Ruby 1.9.
```

Powyzszy kod wymaga kilku komentarzy. Po pierwsze, metody `readline` i `gets` różnią się tylko zachowaniem w momencie napotkania końca pliku. Metoda `gets` wywołana na strumieniu, jeśli napotka koniec pliku, zwraca wartość `nil`. Metoda `readline` w takiej sytuacji zgłasza wyjątek `EOFError`. Jeśli nie wiadomo, z ilu wierszy mogą składać się dane, należy użyć metody `gets`. Jeśli brak kolejnego wiersza oznacza błąd, należy użyć metody `readline`. Do sprawdzenia, czy strumień jest już na końcu pliku, służy metoda `eof?`.

Po drugie, metody `gets` i `readline` niejawnie ustawiają globalną zmienną `$_` na kolejny wiersz tekstu, który zwracają. Zmiennej `$_` używa kilka metod globalnych, na przykład `print`, jeśli nie zostaną do nich przekazane żadne argumenty. W związku z tym powyzszą pętlę `while` można zapisać bardziej zwięźle w następujący sposób:

```
print while DATA.gets
```

Wykorzystanie zmiennej `$_` jest wygodne w krótkich skryptach. Natomiast w długich programach lepszym stylem jest użycie jawnych zmiennych do przechowywania wierszy wczytanych danych.

Po trzecie, metody te są zazwyczaj używane do pracy ze strumieniami tekstowymi (zamiast danych binarnych) i „wiersz danych” jest szeregiem bajtów aż do domyślnego zakończenia wiersza włącznie (na większości platform jest to znak nowego wiersza). Wiersze zwrócone

przez te metody zawierają znak kończący wiersz (mimo że ostatni wiersz w pliku może go nie mieć). Aby usunąć ten znak, należy użyć metody `String.chomp!`. Specjalna zmienna globalna `$/` przechowuje znak końca wiersza. Zmieniając jej wartość, można zmienić domyślne działanie wszystkich metod wczytujących wiersze danych lub przekazać alternatywny separator do dowolnej z tych metod (wliczając iterator `each`). Można to na przykład zrobić podczas odczytywania z pliku pól oddzielonych przecinkami lub pliku binarnego z jakiegoś rodzaju „separatorem rekordów”. Są dwa specjalne przypadki, jeśli chodzi o znak końca wiersza. Jeżeli zostanie on ustawiony na `nil`, metody odczytujące pobierają dane aż do końca strumienia i zwracają całość jako jeden wiersz. Jeśli zostanie ustawiony pusty łańcuch `""`, metody odczytujące będą wczytywać dane akapit po akapicie, jako separator traktując pusty wiersz.

W Ruby 1.9 metody `gets` i `readline` przyjmują opcjonalny argument w postaci liczby całkowitej na pierwszym miejscu lub na drugim, po łańcuchu separatora. Liczba ta określa maksymalną liczbę bajtów, która ma zostać odczytana ze strumienia. Argument ten ma na celu umożliwić uniknięcie przypadkowego wczytania niespodziewanie długich wierszy. Metody te są wyjątkiem od wcześniej sformułowanej reguły. Zwracają zakodowane łańcuchy znaków, mimo że mają argument określający limit wczytywanych bajtów.

W końcu metody wczytujące wiersze `gets`, `readline` i iterator `each` (oraz jego alias `each_line`) pamiętają liczbę wczytanych wierszy. Numer ostatniego wczytanego wiersza można sprawdzić za pomocą metody `lineno`, a do jego ustawiania służy metoda dostępowa `lineno=`. Należy zauważyć, że metoda `lineno` nie sprawdza liczby znaków nowego wiersza w pliku. Zlicza, ile razy zostały wywołane metody wczytujące wiersze, i może zwracać różne wyniki w zależności od użytych znaków oddzielających wiersze:

```
DATA.lineno = 0      # Zaczyna od wiersza 0, mimo że dane znajdują się na końcu pliku.  
DATA.readline        # Wczytuje jeden wiersz danych.  
DATA.lineno          # => 1  
$.                  # => 1: — magiczna zmienna globalna zostaje ustawiona niejawnie.
```

9.7.3.2. Odczytywanie całych plików

Klasa `IO` udostępnia trzy metody klasowe do wczytywania plików bez otwierania żadnego strumienia. Metoda `IO.read` wczytuje cały plik (lub część pliku) i zwraca go jako pojedynczy łańcuch. Metoda `IO.readlines` wczytuje cały plik do tablicy wierszy. Metoda `IO.foreach` iteruje przez wiersze wyznaczonego pliku. W Ruby 1.9 można do tych metod przekazać tablice asocjacyjną, aby określić tryb oraz/lub kodowanie odczytywanego pliku:

```
data = IO.read("data")           # Wczytanie i zwrócenie całego pliku.  
data = IO.read("data", mode:"rb") # Otwarcie w trybie "rb".  
data = IO.read("data", encoding:"binary") # Odczyt niezakodowanych danych.  
data = IO.read("data", 4, 2)       # Wczytanie czterech bajtów, zaczynając od drugiego.  
data = IO.read("data", nil, 6)     # Odczyt od szóstego bajta do końca pliku.  
# Wczytanie wierszy danych do tablicy.  
words = IO.readlines("/usr/share/dict/words")  
# Odczyt po jednym wierszu i inicjacja tablicy asocjacyjnej.  
words = {}  
IO.foreach("/usr/share/dict/words") { |w| words[w] = true}
```

W Ruby 1.9 można użyć metody `IO.copy_stream`, by odczytać plik (lub jego fragment) oraz zapisać jego zawartość do strumienia.

```
IO.copy_stream("/usr/share/dict/words", STDOUT)           # Wyświetlenie słownika.  
IO.copy_stream("/usr/share/dict/words", STDOUT, 10, 100) # Wyświetlenie bajtów 100-109.
```

Mimo iż metody te są zdefiniowane w klasie `IO`, działają na plikach i często można spotkać ich wywołania jako metod klasowych klasy `File` — `File.read`, `File.readlines`, `File.foreach` oraz `File.copy_stream`.

Klasa `IO` udostępnia także metodę egzemplarza o nazwie `read`, która jest podobna do metody klasowej o tej samej nazwie. Bez argumentów wczytuje ona tekst aż do końca strumienia i zwraca go jako zakodowany łańcuch:

```
# Alternatywa dla text = File.read("data.txt").
f = File.open("data.txt")      # Otwarcie pliku.
text = f.read                  # Wczytanie zawartości pliku jako tekst.
f.close                        # Zamknięcie pliku.
```

Metodę egzemplarza `IO.read` można także wywoływać z argumentami, aby wczytać określoną liczbę bajtów ze strumienia. Ten sposób jej użycia został opisany w kolejnym podrozdziale.

9.7.3.3. Wczytywanie bajtów i znaków

Klasa `IO` udostępnia także metody pozwalające wczytywać jeden lub więcej bajtów albo znaków za jednym razem. Metody te znacznie różnią się pomiędzy Ruby 1.8 a 1.9, ponieważ w Ruby 1.9 została zmieniona definicja znaku.

W Ruby 1.8 bajty i znaki są tym samym, a metody `getc` i `readchar` wczytują jeden bajt i zwracają go jako obiekt klasy `Fixnum`. Podobnie jak metoda `gets`, metoda `getc` zwraca wartość `nil` po napotkaniu końca pliku. Natomiast metoda `readchar`, tak jak `readline`, napotykając koniec pliku, zgłasza wyjątek `EOFError`.

W Ruby 1.9 metody `getc` i `readchar` zostały zmodyfikowane. Zamiast obiektu klasy `Fixnum` zwracają łańcuch długości 1. Czytając ze strumienia z kodowaniem wielobajtowym, metody te wczytują tyle bajtów, ile jest potrzebne do wczytania pełnego znaku. Aby w Ruby 1.9 wczytać łańcuch bajt po bajcie, należy użyć nowych metod o nazwach `getbyte` i `readbyte`. Metoda `getbyte` przypomina metody `getc` i `gets` — zwraca `nil` po napotkaniu końca pliku. Metoda `readbyte` natomiast zachowuje się w takiej sytuacji jak metody `readchar` i `readline` — zgłasza wyjątek `EOFError`.

Niektóre programy (na przykład analizatory składni) wczytujące dane ze strumienia po jednym znaku czasami muszą jakiś znak przenieść z powrotem do bufora strumienia, aby został zwrócony przez kolejne wywołanie metody czytającej. Mogą to zrobić za pomocą metody `ungetc`. W Ruby 1.8 metoda ta wymaga obiektu klasy `Fixnum`, a w Ruby 1.9 łańcucha zawierającego jeden znak. Znak wepushnięty z powrotem do bufora zostanie zwrócony przez kolejne wywołanie metody `getc` lub `readchar`:

```
f = File.open("data", "r:binary") # Otwarcie pliku data do odczytu w trybie binarnym.
c = f.getc                      # Wczytanie pierwszego bajtu jako liczby całkowitej.
f.ungetc(c)                      # Wepushnienie bajtu z powrotem.
c = f.readchar                   # Ponowne wczytanie bajtu.
```

Można również iterować i przeglądać znaki oraz bajty strumienia:

```
f.each_byte{|b| ...}           # Iteracja po pozostałych bajtach.
f.bytes                         # Enumerator dla metody each_byte — Ruby 1.9.
f.each_char{|c| ...}            # Iteracja po znakach — Ruby 1.9.
f.chars                          # Enumerator dla metody each_char — Ruby 1.9.
```

Aby wczytać więcej niż jeden bajt za jednym razem, do wyboru jest pięć metod — każda o nieco innym działaniu:

`readbytes(n)`

Wczytuje dokładnie n bajtów i zwraca je jako łańcuch. Blokuje w razie potrzeby, aż zostanie wczytanych n bajtów. Jeśli przed wczytaniem n bajtów nastąpi koniec pliku, zgłasza wyjątek `EOFError`.

`readpartial(n, buffer=nil)`

Wczytuje od 1 do n bajtów i zwraca je jako nowy łańcuch binarny lub, jeśli jako drugi argument zostanie przekazany obiekt klasy `String`, zapisuje je w tym łańcuchu (nadpisując jego dotychczasową zawartość). Jeśli jeden lub więcej bajtów jest dostępnych do odczytu, metoda ta zwraca je (maksymalnie do n) natychmiast. Blokuje tylko, gdy nie są dostępne żadne bajty. Kiedy strumień znajduje się na końcu pliku, metoda ta zgłasza wyjątek `EOFError`.

`read(n=nil, buffer=nil)`

Wczytuje n bajtów (lub mniej, jeśli dojdzie do końca pliku), blokując w razie potrzeby, aż bajty będą gotowe. Bajty są zwarcane w postaci łańcucha binarnego. Jeśli drugi argument jest istniejącym obiektem klasy `String`, bajty zostają zapisane w tym obiekcie (zastępując jego aktualną treść) i zwracany jest ten łańcuch. Jeśli strumień znajduje się na końcu pliku i została określona wartość n , metoda ta zwraca wartość `nil`. Jeśli metoda ta zostanie wywołana na końcu pliku, a argument n zostanie pominięty lub ma wartość `nil`, zwraca pusty łańcuch `" "`.

Jeśli argument n ma wartość `nil` lub zostanie opuszczony, metoda ta wczytuje resztę strumienia i zwraca ją jako zakodowany łańcuch znaków, a nie niezakodowany łańcuch bajtów.

`read_noblock(n, buffer=nil)`

Wczytuje bajty (maksymalnie n), które są aktualnie dostępne, i zwraca je jako łańcuch przy użyciu łańcucha `buffer`, jeśli zostanie podany. Metoda ta nie blokuje. Jeśli nie ma żadnych danych gotowych do odczytu ze strumienia (może się to zdarzyć w gnieździe sieciowym lub wejściu standardowym), metoda ta zgłasza wyjątek `SystemCallError`. Wywołanie na końcu pliku powoduje zgłoszenie wyjątku `EOFError`.

Metoda ta jest nowością w Ruby 1.9 (Ruby 1.9 udostępnia także inne nieblokujące metody `I/O`, ale są to metody niskiego poziomu i nie opisujemy ich tutaj).

`sysread(n)`

Niniejsza metoda działa podobnie jak metoda `readbytes`, ale działa na niższym poziomie bez buforowania. Nie należy mieszać tej metody z jakimkolwiek innymi metodami wczytującymi wiersze lub bajty, ponieważ są one ze sobą niekompatybilne.

Poniżej znajduje się przykładowy kod, który można wykorzystać do wczytania pliku binarnego:

```
f = File.open("data.bin", "rb:binary") # Bez konwersji znaków nowego wiersza, bez kodowania.  
magic = f.readbytes(4) # Cztery pierwsze bajty identyfikują typ pliku.  
exit unless magic == "INTS" # Liczba magiczna INTS (ASCII).  
bytes = f.read # Wczytanie reszty pliku.  
# Kodowanie jest binarne, a więc jest to łańcuch bajtów.  
data = bytes.unpack("i*") # Konwersja bajtów na tablicę liczb całkowitych.
```

9.7.4. Zapis w strumieniu

Metody klasy `IO` służące do zapisu w strumieniach są zwierciadlanym odbiciem metod odczytu. Strumienie `STDOUT` i `STDERR` umożliwiają zapisywanie danych, podobnie jak pliki otwarte w innym trybie niż `r` lub `rb`.

Klasa `IO` udostępnia metodę `putc` do zapisu pojedynczych bajtów lub znaków w strumieniu. Metoda ta przyjmuje jako argument bajt lub łańcuch zawierający jeden znak. Jej działanie jest takie samo w Ruby 1.8, jak i Ruby 1.9:

```
o = STDOUT
# Dane wyjściowe w postaci jednego znaku.
o.putc(65)          # Zapis jednego bajta 65 (wielka litera A).
o.putc("B")         # Zapis jednego bajta 66 (wielka litera B).
o.putc("CD")        # Zapis tylko pierwszego bajta łańcucha.
```

Klasa `IO` udostępnia także kilka innych metod do zapisu dowolnych łańcuchów. Metody te różnią się od siebie liczbą przyjmowanych argumentów i tym, czy dodawanie są znaki końca wiersza, czy nie. Przypomnijmy, że w Ruby 1.9 wyjściowe dane tekstowe są transkodowane na zewnętrzne kodowanie strumienia, jeśli zostało ono określone:

```
o = STDOUT
# Wysyłanie łańcucha na wyjście.
o << x           # Wysłanie x.to_s.
o << x << y      # Można tworzyć łańcuchy — output x.to_s + y.to_s.
o.print            # Wysłanie $_ + $|.
o.print s          # Wysłanie s.to_s + $|.
o.print s,t        # Wysłanie s.to_s + t.to_s + $|.
o.printf fmt,*args # Wysyla fmt%[args].
o.puts             # Wysłanie znaku nowego wiersza.
o.puts x           # Wysłanie x.to_s.chomp plus znak nowego wiersza.
o.puts x,y         # Wysłanie x.to_s.chomp, znak nowego wiersza, y.to_s.chomp, znak nowego wiersza.
o.puts [x,y]        # To samo co powyżej.
o.write s          # Wysłanie s.to_s, zwrócenie s.to_s.length.
o.syswrite s       # Niskopoziomowa wersja metody write.
```

Do strumieni wyjściowych można dodawać dane, jak do łańcuchów i tablic oraz można do nich zapisywać dane za pomocą operatora `<<`. Metoda `puts` jest jedną z najczęściej używanych metod wyjściowych. Konwertuje każdy swój argument na łańcuch i zapisuje go w strumieniu. Jeśli łańcuch nie ma jeszcze na końcu znaku nowego wiersza, dodaje go. Jeśli którykolwiek z argumentów metody `puts` jest tablicą, jest ona rekursywnie rozwijana i każdy jej element jest drukowany we własnym wierszu, tak jakby został przekazany bezpośrednio jako argument do metody `puts`. Metoda `print` konwertuje swoje argumenty na łańcuchy i wysyła je do strumienia. Jeśli domyślna wartość `nil` globalnego separatora pól `$` została zmieniona na coś innego, wartość ta jest wysyłana pomiędzy każdą parą argumentów metody `print`. Jeśli domyślna wartość `nil` globalnego separatora rekordów `$/` została zmieniona na coś innego, wartość ta jest wysyłana po wydrukowaniu wszystkich argumentów

Metoda `printf` jako pierwszy argument wymaga łańcucha formatu i interpoluje wartości wszystkich dodatkowych argumentów do tego łańcucha przy użyciu operatora `String%`. Następnie wysyła na wyjście ten łańcuch bez separatora wierszy lub rekordów.

Metoda `write` wysyła na wyjście swój pojedynczy argument jak operator `<<` i zwraca liczbę zapisanych bajtów. W końcu metoda `syswrite` jest niskopoziomową, niebuforowaną i nie-wykonującą transkodowania wersją metody `write`. Metoda `syswrite` powinna być używana osobno — nie należy jej mieszać z innymi metodami zapisu.

9.7.5. Metody dostępu swobodnego

Niektóre strumienie, na przykład reprezentujące gniazda sieciowe lub dane wprowadzane przez użytkownika w konsoli, są strumieniami sekwencyjnymi — po wczytaniu z nich danych lub zapisaniu danych do nich nie można już się cofnąć. Inne strumienie, na przykład odczytujące z lub zapisujące do plików lub łańcuchów, pozwalają na swobodny dostęp przy użyciu opisanych tutaj metod. Użycie tych metod na strumieniu nieumożliwiającym dostępu swobodnego spowoduje zgłoszenie wyjątku `SystemCallException`:

```
f = File.open("test.txt")
f.pos          #=> 0 — zwrócenie aktualnej pozycji w bajtach.
f.pos = 10    # Przejście do pozycji 10.
f.tell        #=> 10 — synonim metody pos.
f.rewind      # Przejście z powrotem do pozycji 0, reset numeru wiersza do 0.
f.seek(10, IO::SEEK_SET) # Przejście do bezwzględnej pozycji 10.
f.seek(10, IO::SEEK_CUR) # Przejście o 10 bajtów dalej od aktualnej pozycji.
f.seek(-10, IO::SEEK_END) # Przejście do pozycji oddalonej o 10 bajtów od końca.
f.seek(0, IO::SEEK_END)   # Przejście na sam koniec pliku.
f.eof?           #=> true — jesteś na końcu.
```

Jeśli w programie zostały użyte metody `sysread` lub `syswrite`, to zamiast metody `seek` należy użyć metody `sysseek`. Metoda ta działa bardzo podobnie do metody `seek`, z tą różnicą, że po każdym wywołaniu zwraca nową pozycję w pliku:

```
pos = f.sysseek(0, IO::SEEK_CUR)  # Sprawdzenie aktualnej pozycji.
f.sysseek(0, IO::SEEK_SET)       # Przewinięcie strumienia do tyłu.
f.sysseek(pos, IO::SEEK_SET)    # Powrót do pierwotnej pozycji.
```

9.7.6. Zamykanie, opróżnianie i testowanie strumieni

Po zakończeniu odczytu z lub zapisu do strumienia należy go zamknąć za pomocą metody `close`. Powoduje to opróżnienie bufora zawierającego dane wejściowe lub wyjściowe i zwolenie zasobów systemu operacyjnego. Niektóre metody otwierające strumienie pozwalają na wiązanie z nimi bloków. Przekazują otwarty strumień do bloku i automatycznie zamykają go, kiedy blok zakończy działanie. Taka obsługa strumieni daje pewność, że zostaną one zamknięte nawet wówczas, gdy wystąpi wyjątek:

```
File.open("test.txt") do |f|
  # Użycie strumienia f.
  # Wartość tego bloku staje się wartością zwracaną metodą open.
end # Strumień f zostaje zamknięty automatycznie w tym miejscu.
```

Alternatywą dla bloku może być klauzula `ensure`:

```
begin
  f = File.open("test.txt")
  # Użycie strumienia f.
ensure
  f.close if f
end
```

Gniazda sieciowe są zaimplementowane jako obiekty klasy `IO` z osobnymi wewnętrznymi strumieniami odczytu i zapisu. Do zamykania tych wewnętrznych strumieni służą metody `close_read` i `close_write`. Mimo że pliki można otwierać do odczytu i zapisu równocześnie, nie można na tych obiektach klasy `IO` używać metod `close_read` i `close_write`.

Metody wyjściowe w języku Ruby (z wyjątkiem metody `syswrite`) buforują dane wyjściowe w celu zwiększenia wydajności. Bufor wyjściowy jest opróżniany co określony czas, na przykład po wysłaniu znaku nowego wiersza lub wczytaniu danych z odpowiedniego strumienia

wejściowego. Czasami jednak konieczne jest wymuszenie wysłania danych natychmiast po przez opróżnienie bufora:

```
out.print 'wait>' # Wyświetlenie znaku zachęty.  
out.flush          # Ręczne opróżnienie bufora wyjściowego do systemu operacyjnego.  
sleep(1)           # Znak zachęty pojawia się przed zaśnięciem.  
out.sync = true    # Automatyczne opróżnianie bufora po każdym zapisie.  
out.sync = false   # Wyłączenie automatycznego opróżniania.  
out.sync           # Zwraca bieżący tryb synchronizacji.  
out.fsync          # Opróżnia bufor wyjściowy i prosi system operacyjny o opróżnienie buforów.  
                   # Zwraca wartość nil, jeśli nie jest obsługiwana na bieżącej platformie.
```

Klasa IO udostępnia kilka predykatów pozwalających sprawdzić stan strumienia:

```
f.eof?            # true, jeśli strumień jest na końcu pliku.  
f.closed?        # true, jeśli strumień został zamknięty.  
f.tty?           # true, jeśli strumień jest interaktywny.
```

Jedyna z powyższych metod, która wymaga dodatkowych wyjaśnień, to tty?. Metoda ta i jej alias isatty (bez znaku zapytania) zwraca wartość true, jeśli strumień jest połączony z interaktywnym urządzeniem, jak okno terminala lub klawiatura, przy którym siedzi człowiek. Jeśli strumień nie jest interaktywny (jest to na przykład plik, potok lub gniazdo) metody te zwracają wartość false. W programie można wykorzystać metodę tty?, aby uniknąć wyświetlenia zachęty dla użytkownika do podawania danych, kiedy standardowy strumień wejściowy został przekierowany i przychodzi na przykład z pliku.

9.8. Programowanie sieciowe

Funkcje programowania sieciowego języka Ruby umiejscowione są w bibliotece standardowej, a nie w rdzennych klasach. Dlatego kolejne podrozdziały nie są próbą wyliczenia wszystkich dostępnych klas lub metod. Zamiast tego pokazują na prostych przykładach, jak wykonać najczęściej spotykane zadania związane z programowaniem sieciowym. Więcej szczegółowych informacji można uzyskać przy użyciu narzędzia ri.

Na najniższym poziomie połączenia sieciowe są obsługiwane przez gniazda będące rodzajem obiektów klasy IO. Mając otwarte gniazdo, można odczytywać dane z innego komputera i zapisywać je do niego, jakby odczytywało się je z pliku lub zapisywało do niego. Hierarchia klas gniazd jest nieco niejasna, ale w przedstawionych dalej przykładach szczegóły nie są ważne. Klienci internetowi używają klasy TCPSocket, a serwery internetowe klasy TCPServer (która także jest gniazdem). Wszystkie klasy gniazd należą do biblioteki standardowej, a więc aby ich użyć, należy w kodzie programu wpisać następujący wiersz:

```
require 'socket'
```

9.8.1. Bardzo prosty klient

Do pisania internetowych aplikacji klienckich służy klasa TCPSocket. Aby utworzyć obiekt tej klasy, należy wywołać metodę klasową TCPSocket.open lub jej synonim TCPSocket.new. Jako pierwszy argument należy przekazać nazwę hosta, z którym ma zostać nawiązane połączenie, a drugi numer portu. Numer portu powinien być liczbą całkowitą z przedziału 1-65535 w postaci obiektu klasy Fixnum lub String. Różne protokoły internetowe używają różnych portów. Na przykład serwery sieciowe domyślnie korzystają z portu numer 80. Zamiast numeru portu można także przekazać nazwę usługi sieciowej, na przykład http, w postaci łańcucha. Nie jest to jednak dobrze udokumentowane i może zależeć od systemu.

Po otwarciu gniazda można z niego odczytywać dane tak jak z każdego innego obiektu klasy IO. Po zakończeniu operacji należy pamiętać o zamknięciu gniazda, tak jak zamyka się plik. Poniżej przedstawiony jest kod źródłowy prostego klienta łączącego się z podanym hostem i portem, odczytującego z gniazda wszystkie dostępne dane i kończącego działanie:

```
require 'socket'          # Gniazda znajdują się w bibliotece standardowej.  
host, port = ARGV        # Host i port z wiersza poleceń.  
s = TCPSocket.open(host, port) # Otwarcie gniazda do hosta i portu.  
while line = s.gets       # Odczytanie danych z gniazda.  
    puts line.chop        # Wydruk danych ze znakiem końca wiersza platformy.  
end  
s.close                  # Zamknięcie gniazda po skończeniu pracy.
```

Metodę `TCPSocket.open`, podobnie jak `File.open`, można wywołać z blokiem. W takim przypadku metoda ta przekazuje otwarte gniazdo do bloku i automatycznie zamyka je po zwróceniu przez ten blok wartości. W związku z tym niniejszy kod można również zapisać tak:

```
require 'socket'  
host, port = ARGV  
TCPSocket.open(host, port) do |s| # Blokowa wersja metody open.  
    while line = s.gets  
        puts line.chop  
    end  
end                                # Gniazdo zostaje zamknięte automatycznie.
```

Ten klient nadaje się do pracy z usługami w starym stylu (obecnie niedziałającymi), jak usługa *daytime* systemu Unix. W przypadku takich usług klient nie wysyła żadnych żądań, tylko łączy się, a serwer przesyła odpowiedź. Jeśli nie masz dostępu do hosta internetowego z uruchomionym serwerem, nie panikuj — kolejny podrozdział opisuje tworzenie równie prostego serwera czasu.

9.8.2. Bardzo prosty serwer

Do pisania serwerów internetowych służy klasa `TCPServer`. Zasadniczo obiekt klasy `TCPServer` jest fabryką obiektów klasy `TCPSocket`. Aby wyznaczyć numer portu dla swojej usługi i utworzyć obiekt klasy `TCPServer`, należy wywołać metodę `TCPServer.open`. Następnie należy wywołać metodę `accept` zwróconego obiektu `TCPServer`. Metoda ta czeka, aż klient połączy się z podanym portem, i zwraca obiekt klasy `TCPSocket` reprezentujący to połączenie.

Poniżej przedstawiamy przykładowy prosty serwer czasu nasłuchujący połączeń na porcie 2000. Kiedy klient połączy się z tym portem, serwer wysyła do niego aktualny czas i zamyka gniazdo, zamykając tym samym połączenie z klientem:

```
require 'socket'          # Gniazda z biblioteki standardowej.  
server = TCPserver.open(2000) # Gniazdo nasłuchujące na porcie 2000.  
loop {  
    client = server.accept  # Nieskończona pętla — serwery działają bez przerwy.  
    client.puts(Time.now.ctime) # Oczekiwanie na połączenie klienta.  
    client.close             # Wysłanie informacji o czasie do klienta.  
    client.close             # Rozłączenie z klientem.  
}
```

Aby przetestować ten kod, uruchom go w tle lub innym oknie terminala. Następnie uruchom klienta, którego kod został przedstawiony wcześniej za pomocą poniższego polecenia:

```
ruby client.rb localhost 2000
```

9.8.3. Datagramy

Większość protokołów internetowych jest zaimplementowana przy użyciu klas `TCPSocket` i `TCPServer`, jak pokazano wcześniej. Alternatywą o mniejszym narzucie są datagramy UDP będące obiektami klasy `UDPSocket`. Protokół UDP pozwala komputerom wysyłać pojedyncze pakiety danych do innych komputerów bez opóźnienia związanego z ustanawianiem trwałego połączenia. W poniższym kodzie klienta i serwera klient wysyła datagram zawiązający łańcuch tekstu do określonego hosta i portu. Serwer, który powinien działać na tym hoście i nasłuchiwać na tym porcie, odbiera ten tekst, konwertuje go na wielkie litery (wiem, to nie jest zbyt ciekawa usługa) i wysyła go z powrotem w drugim datagramie.

Pierwszy jest kod klienta. Zauważ, że mimo iż obiekty klasy `UDPSocket` są obiektami klasy `IO`, datagramy znacznie różnią się od innych strumieni `IO`. Z tego powodu unikaj metod klasy `IO` i używaj niższego poziomu metod wysyłających i odbierających dane klasy `UDPSocket`. Drugi argument klasy metody `send` określa znaczniki. Jest on wymagany, nawet jeśli nie są określone żadne znaczniki. Argument metody `recvfrom` określa maksymalną ilość danych, które mogą zostać odebrane. W tym przypadku klient i serwer zostali ograniczeni do transferu jednego kilobajta:

```
require 'socket'  
host, port, request = ARGV  
ds = UDPSocket.new  
ds.connect(host, port)  
ds.send(request, 0)  
response,address = ds.recvfrom(1024) # Oczekiwanie na odpowiedź (maksymalnie 1 kb).  
puts response # Wydruk odpowiedzi.
```

W kodzie serwera wykorzystana została klasa `UDPSocket`, tak samo jak w kodzie klienta — nie istnieje żadna specjalna klasa `UDPServer` do tworzenia serwerów datagramowych. Za- miast wywoływać metodę `connect`, aby połączyć się z gniazdem, wywołana została metoda `bind` informująca gniazdo, na którym porcie ma nasłuchiwać. Następnie serwer używa metod `send` i `recvfrom`, tak jak klient, ale w odwrotnej kolejności. Metodę `recvfrom` wywołuje, aby poczekać do odebrania datagramu na określonym porcie. Kiedy to się stanie, konwertuje odebrany tekst na wielkie litery i wysyła go z powrotem. Ważne jest, aby zauważyc, że metoda `recvfrom` zwraca dwie wartości. Pierwsza z nich to odebrane dane. Druga jest tablicą zawierającą informacje o pochodzeniu tych danych. Z tablicy tej pobierane są dane na temat hosta i portu i na ich podstawie odpowiedź jest wysyłana do klienta:

```
require 'socket'  
port = ARGV[0]  
ds = UDPSocket.new  
ds.bind(nil, port)  
loop do  
    request,address=ds.recvfrom(1024) # Oczekiwanie na jakieś dane.  
    response = request.upcase # Konwersja tekstu na wielkie litery.  
    clientaddr = address[3] # Adres IP, z którego przyszło żądanie.  
    clientname = address[2] # Nazwa hosta.  
    clientport = address[1] # Numer portu, z którego żądanie zostało wysłane.  
    ds.send(response, 0, clientaddr, clientport) # Wysłanie odpowiedzi tam...  
    # ... skąd przyszło żądanie.  
    # Zapisanie połączenia z klientem w dzienniku.  
    puts "Połączenie z: #{clientname} #{clientaddr} #{clientport}"  
end
```

9.8.4. Bardziej skomplikowany klient

Poniższy kod przedstawia bardziej rozwiniętego klienta internetowego w stylu telnetu. Łączy się on z określonym hostem i portem, a następnie za pomocą pętli odczytuje jeden wiersz danych z konsoli, wysyła je do serwera oraz odczytuje i drukuje odpowiedź serwera. Demonstruje sposób określania lokalnych i zdalnych adresów połączenia sieciowego, dodano w nim obsługę błędów i wykorzystano metody klasy `IO.read_nonblock` i `readpartial`, które zostały opisane wcześniej. Kod zawiera dużo komentarzy i jego zrozumienie nie powinno być trudne:

```
require 'socket'          # Gniazda z biblioteki standardowej.
host, port = ARGV         # Nazwa hosta i numer portu z wiersza poleceń.
begin                     # Dla obsługi wyjątków.
  # Podczas łączenia będzie wyświetlane odpowiednia informacja.
  STDOUT.print "łączenie..."           # Co się dzieje.
  STDOUT.flush                      # Pokazanie tego natychmiast.
  s = TCPSocket.open(host, port)      # Łączenie.
  STDOUT.puts "ukończono"            # Informacja o zakończeniu operacji.
  # Wyświetlenie informacji o połączeniu.
  local, peer = s.addr, s.peeraddr
  STDOUT.print "Połączono z #{peer[2]}:#{peer[1]}"
  STDOUT.puts " przy użyciu portu lokalnego #{local[1]}"
  # Odczekanie chwilę, aby sprawdzić, czy serwer przyśle jakiś komunikat początkowy.
  begin
    sleep(0.5)                   # Odczekanie pół sekundy.
    msg = s.read_nonblock(4096)     # Odczytanie tego, co jest gotowe.
    STDOUT.puts msg.chop          # Wyświetlenie tego.
  rescue SystemCallError
    # Jeśli nic nie było gotowe do odczytu, wyjątek zostanie zignorowany.
  end
  # Początek pętli interakcji klienta z serwerem.
  loop do
    STDOUT.print '> '           # Wyświetlenie znaku zachęty do podania danych lokalnych.
    STDOUT.flush                 # Uwidocznienie znaku zachęty.
    local = STDIN.gets           # Odczytanie jednego wiersza tekstu z konsoli.
    break if !local             # Zakończenie, jeśli w konsoli nie ma żadnych danych.
    s.puts(local)               # Wysłanie wczytanego wiersza do serwera.
    s.flush                     # Wymuszenie wyjścia tego wiersza.
    # Odczytanie odpowiedzi serwera i jej wydrukowanie.
    # Serwer może przysłać więcej niż jeden wiersz tekstu, a więc została użyta metoda readpartial
    # odczytywająca, co zostanie przesłane (jeśli dane te przyjdą w jednym kawałku).
    response = s.readpartial(4096) # Odczytanie odpowiedzi serwera.
    puts(response.chop)         # Wyświetlenie tej odpowiedzi.
  end
  rescue
    puts $!                    # Jeśli coś pójdzie nie tak,
  ensure
    s.close if s               # nie można zapomnieć zamknąć gniazda.
  end
```

9.8.5. Serwer multipleksujący

Prosty serwer czasu przedstawiony wcześniej nie utrzymywał połączenia z żadnym klientem — informował tylko klienta o aktualnym czasie i rozłączał się. Bardziej zaawansowane serwery podtrzymują połączenie i aby był z nich jakiś pożytek, muszą łączyć się i odpowiadać na żądania wielu klientów naraz. Jednym ze sposobów na osiągnięcie tego jest użycie wątków — każdy klient działałby we własnym wątku. Przykładowy serwer wielowątkowy przedstawiamy w dalszej części tego rozdziału. W tej części zajmiemy się budową serwera multipleksującego przy użyciu metody `Kernel.select`.

Jeśli z serwerem jest połączonych kilku klientów, nie może on wywołać na rzecz gniazda żadnego z nich blokującej metody typu `gets`. Jeśli założy blokadę w oczekiwaniu na dane od jednego klienta, nie będzie mógł odebrać danych od innych klientów ani połączyć się z żadnymi innymi klientami. Rozwiązaniem tego problemu jest metoda `select`. Pozwala ona zablokować całą tablicę obiektów klasy `IO` i zwraca informację, kiedy na którymś z tych obiektów pojawią się jakaś aktywność. Wartością zwrotną metody `select` jest tablica tablic obiektów klasy `IO`. Pierwszym elementem tej tablicy jest tablica strumieni (w tym przypadku gniazd) zawierających dane do odczytu (lub połączenie do zaakceptowania).

Dysponując tymi informacjami o metodzie `select`, nikt nie powinien mieć problemów ze zrozumieniem poniższego kodu serwera. Usługa świadczona przez ten serwer jest prosta — odwraca każdy wiersz danych od klienta i zwraca go. Interesujący jest w tym przypadku mechanizm obsługujący wiele połączeń. Należy zauważać, że za pomocą metody `select` monitorowane są zarówno obiekty klasy `TCPServer`, jak i klasy `TCPSocket`. Ponadto warto zauważać, że serwer ten obsługuje przypadek zarówno rozłączenia na prośbę klienta, jak i rozłączenie klienta w niespodziewany sposób:

```
# Niniejszy serwer wczytuje wiersz danych od klienta, odwraca go
# i wysyła z powrotem. Jeśli klient wyśle laicuch "koniec",
# połączenie zostaje zakończone. Do obsługi sesji została użyta metoda Kernel.select.
require 'socket'
server = TCPServer.open(2000) # Nasłuchiwanie na porcie 2000.
sockets = [server] # Tablica gniazd, które będą monitorowane.
log = STDOUT # Wysyłanie komunikatów dziennika na wyjście standardowe.
while true
  ready = select(sockets) # Oczekiwanie, aż gniazdo będzie gotowe.
  readable = ready[0] # Z tych gniazd można odczytywać dane.
  readable.each do |socket|
    if socket == server # Iteracja przez gniazda umożliwiające odczyt.
      client = server.accept # Jeśli gniazdo serwera jest gotowe,
      sockets << client # akceptuje nowego klienta.
      log.puts "# Dodanie go do zestawu gniazd.
      # Poinformowanie klienta, z czym i gdzie się połączył.
      client.puts "Usługa odwracania v0.01 działa na #{Socket.gethostname}""
      # Zarejestrowanie faktu połączenia z klientem.
      log.puts "Przyjęto informację od #{client.peeraddr[2]}"" # W przeciwnym przypadku klient jest gotowy.
    else
      input = socket.gets # Wczytanie danych od klienta.
      # Jeśli brak danych, klient rozłączył się.
      if !input
        log.puts "Klient #{socket.peeraddr[2]} rozłączył się."
        sockets.delete(socket) # Zaprzestanie monitorowania tego gniazda.
        socket.close # Zamknięcie go.
        next # Przejście do następnego.
      end
      input.chop! # Przycięcie danych od klienta.
      if (input == "koniec") # Jeśli klient prosi o zakończenie.
        socket.puts("Do widzenia!"); # mówi mu do widzenia.
        log.puts "Closing connection to #{socket.peeraddr[2]}"" # Zakończenie monitorowania gniazda.
        sockets.delete(socket) # Zakończenie sesji.
        socket.close # Zakończenie.
      else
        socket.puts(input.reverse) # W przeciwnym przypadku klient nie skończył.
        # Odwracasz dane i wysydas je z powrotem.
      end
    end
  end
end
```

9.8.6. Ładowanie stron internetowych

Z pomocą biblioteki gniazd można zaimplementować każdy protokół internetowy. Poniżej znajduje się przykładowy kod ładujący zawartość strony internetowej:

```
require 'socket'          # Gniazda.  
  
host = 'www.example.com'  # Serwer sieciowy.  
port = 80                 # Domyślny port HTTP.  
path = "/index.html"       # Wymagany plik.  
# Żądanie HTTP wysypane, aby załadować plik.  
request = "GET #{path} HTTP/1.0\r\n\r\n"  
socket = TCPSocket.open(host, port)  # Połączenie z serwerem.  
socket.print(request)           # Wysłanie żądania.  
response = socket.read          # Odczytanie pełnej odpowiedzi.  
# Podzielenie odpowiedzi według pierwszego pustego wiersza na część nagłówkową i ciało.  
headers, body = response.split("\r\n\r\n", 2)  
print body                     # Wyświetlenie ciała.
```

HTTP to skomplikowany protokół. Powyższy kod potrafi obsłużyć tylko proste przypadki. Do pracy z tym protokołem można użyć wbudowanej biblioteki `Net::HTTP`. Poniższy kod działa tak samo jak powyższy:

```
require 'net/http'          # Potrzebna biblioteka.  
host = 'www.example.com'    # Serwer sieciowy.  
path = '/index.html'        # Wymagany plik.  
http = Net::HTTP.new(host)   # Utworzenie połączenia.  
headers, body = http.get(path) # Żądanie pliku.  
if headers.code == "200"      # Sprawdzenie kodu stanu.  
    # UWAGA: kod nie jest liczbą!  
    print body                # Wydrukowanie ciała, jeśli udało się je pobrać.  
else  
    puts "#{headers.code} #{headers.message}" # wyświetlasz komunikat o błędzie.  
end
```

Podobne biblioteki dostępne są dla protokołów FTP, SMTP, POP i IMAP. Szczegóły na temat tych standardowych bibliotek wykraczają poza zakres tematyczny tej książki.

Na koniec przypomnijmy, że biblioteka `open-uri`, którą opisywaliśmy wcześniej, jeszcze bardziej ułatwia ładowanie stron internetowych:

```
require 'open-uri'  
open("http://www.example.com/index.html") {|f|  
    puts f.read  
}
```

9.9. Wątki i współbieżność

Tradycyjne programy działają w jednym wątku wykonawczym — instrukcje składające się na program są wykonywane sekwencyjnie aż do ostatniej. Program **wielowątkowy** działa w więcej niż jednym wątku wykonawczym. Instrukcje w każdym z tych wątków są wykonywane sekwencyjnie, ale same wątki mogą działać równolegle z innymi wątkami — na przykład w procesorze wielordzeniowym. Często jest tak, że wątki nie są wykonywane w rzeczywistości równolegle (na przykład w komputerach z jednym jednordzeniowym procesorem). Równoległość jest symulowana poprzez przeplatanie wykonywania poszczególnych wątków.

Programy wykonujące dużo obliczeń, na przykład oprogramowanie do obróbki grafiki, nazywane są czasami programami **zachłannymi**. Mogą one skorzystać na wielowątkowości tylko w obecności wielu procesorów wykonujących poszczególne obliczenia równocześnie. Jednak większość programów nie jest tak bardzo zachłanna. Wiele z nich, na przykład przeglądarki internetowe, spędzają większość czasu na oczekiwaniu na pliki czy połączenia sieciowe. Tego typu programy ograniczone są operacjami wejścia i wyjścia, a ich angielska nazwa to *IO-bound*. Programy tego typu mogą skorzystać na wielowątkowości nawet przy tylko jednym procesorze. Przeglądarka internetowa może w jednym wątku wyświetlić obraz, podczas gdy inny wątek czeka na kolejny obraz z sieci.

Pisanie programów wielowątkowych w Ruby ułatwia klasa `Thread`. Aby uruchomić nowy wątek, wystarczy związać blok kodu z wywołaniem metody `Thread.new`. Zostanie utworzony nowy wątek do wykonania tego bloku kodu, a oryginalny wątek zwróci natychmiast z metody `Thread.new` i wznowi wykonywanie od kolejnej instrukcji:

```
# Pierwszy wątek.  
Thread.new {  
    # Ten kod wykonywany jest przez drugi wątek.  
}  
# Ten kod wykonywany jest przez pierwszy wątek.
```

Opis wątków w języku Ruby zaczniemy od objaśnienia modelu i API wątków. Wstępne podrozdziały opisują, co to jest cykl życia wątku, planowanie wykonywania wątków oraz stany wątków. Następnie po zapoznaniu się z podstawami przejdziemy do analizy przykładowego kodu i bardziej zaawansowanych tematów, jak synchronizacja wątków.

Na koniec należy jeszcze zauważyc, że programy Ruby mogą osiągnąć współbieżność na poziomie procesu systemu operacyjnego poprzez wykonywanie zewnętrznego kodu wykonywalnego lub odgałęzianie nowych kopii interpretera Ruby. Działania te są jednak zależne od systemu operacyjnego i opisane zostały krótko w rozdziale 10. Więcej informacji na ten temat można uzyskać za pomocą narzędzi `ri` w metodach `Kernel.system`, `Kernel.exec`, `Kernel. \rightarrow fork`, `IO.popen` i module `Process`.

Wątki a zależność od platformy

Różne systemy operacyjne różnie implementują wątki. Dodatkowo różne implementacje języka Ruby w różny sposób nakładają warstwę wątków na wątki systemu operacyjnego. Na przykład standardowa implementacja C Ruby 1.8 używa jednego natywnego wątku i wykonyuje wszystkie wątki w tym jednym wątku natywnym. Oznacza to, że w Ruby 1.8 wątki są bardzo lekkie, ale nigdy nie działają równolegle, nawet w systemach wieloprocesorowych.

W Ruby 1.9 wprowadzono zmiany — dla każdego wątku alokowany jest natywny wątek. Ponieważ jednak niektóre biblioteki C użyte w tej implementacji same nie są bezpieczne wątkowo, Ruby 1.9 jest bardzo konserwatywny i nigdy nie pozwala więcej niż jednemu natywnemu wątkowi działać w jednym czasie (to ograniczenie może zostać rozluźnione w późniejszych wydaniach Ruby 1.9, jeśli kod C uda się przerobić na bezpieczny wątkowy).

Implementacja w Javie języka Ruby o nazwie JRuby rzuca każdy wątek Ruby na wątek Javy. Jednak implementacja i zachowanie wątków Javy zależy z kolei od implementacji maszyny wirtualnej Javy. Najnowsze implementacje Javy zazwyczaj implementują wątki jako natywne i pozwalają na prawdziwe równoległe wykonywanie ich na kilku procesorach.

9.9.1. Cykl życia wątku

Jak napisaliśmy wcześniej, do tworzenia nowych wątków służy metoda `Thread.new`. Można też użyć jednego z jej synonimów — `Thread.start` lub `Thread.fork`. Po utworzeniu wątku nie trzeba go uruchamiać. Zostaje on uruchomiony automatycznie, gdy zwolnią się zasoby procesora. Wartością zwracaną metody `Thread.new` jest obiekt klasy `Thread`. Klasa `Thread` udostępnia kilka metod do badania i manipulowania wątkiem podczas jego działania.

Wątek wykonuje kod znajdujący się w bloku związanym z wywołaniem metody `Thread.new` i zatrzymuje się. Wartość ostatniego wyrażenia w tym bloku jest wartością zwracaną wątku. Można ją sprawdzić za pomocą metody `value` obiektu klasy `Thread`. Jeśli wątek został wykonany w całości, metoda `value` zwraca jego wartość od razu. W przeciwnym przypadku metoda ta blokuje i nie zwraca wartości, dopóki wątek nie zakończy działania.

Metoda klasowa `Thread.current` zwraca obiekt klasy `Thread` reprezentujący aktualny wątek. To pozwala wątkom na manipulację sobą. Metoda klasowa `Thread.main` zwraca obiekt klasy `Thread` reprezentujący wątek główny — początkowy wątek wykonawczy, który rozpoczął się w chwili uruchomienia programu.

9.9.1.1. Wątek główny

Wątek główny jest wyjątkowy — interpreter Ruby przestaje działać, kiedy wątek ten zakończy działanie. Robi to nawet, jeśli inne wątki utworzone w wątku głównym jeszcze nie skończyły działania. Dlatego trzeba uważać, aby wątek główny nigdy nie kończył się przed zakończeniem innych wątków. Jednym ze sposobów na rozwiązywanie tego problemu jest napisanie wątku głównego w postaci nieskończonej pętli. Innym sposobem jest poczekanie, aż wszystkie interesujące programistę wątki zakończą się. Napisaliśmy wcześniej, że metodę `value` wątku można wywołać, aby poczekać na jego zakończenie. Jeśli wartość wątków nie jest ważna, można do tego celu użyć metody `join`.

Poniższa metoda czeka, aż wszystkie wątki poza wątkami głównym i bieżącym (co czasami jest tym samym) zakończą działanie:

```
# Czeka, aż wszystkie wątki (poza bieżącym i głównym) zakończą działanie.  
# Zakłada, że podczas oczekiwania nie zostaną uruchomione żadne dodatkowe wątki.  
def join_all  
  main = Thread.main          # Wątek główny.  
  current = Thread.current    # Bieżący wątek.  
  all = Thread.list           # Wszystkie wątki jeszcze działają.  
  # Wywołanie metody join na rzecz każdego wątku.  
  all.each { |t| t.join unless t == current or t == main }  
end
```

9.9.1.2. Wątki a nieobsłużone wyjątki

Jeśli w wątku głównym zostanie zgłoszony wyjątek, który nie jest nigdzie obsługiwany, interpreter Ruby wydrukuje komunikat i zakończy działanie. W wątkach innych niż główny nieobsłużone wyjątki powodują zatrzymanie wątku. Domyślnie jednak nie powodują wydrukowania przez interpreter komunikatu ani zakończenia jego działania. Jeśli wątek `t` zakończy działanie z powodu nieobsłużonego wyjątku, a inny wątek `s` wywoła metodę `t.join` lub `t.value`, wyjątek zgłoszony w wątku `t` zostanie zgłoszony w wątku `s`.

Aby każdy nieobsłużony wyjątek w każdym wątku powodował zamknięcie interpretera, należy użyć metody klasowej `Thread.abort_on_exception`:

```
Thread.abort_on_exception = true
```

Aby nieobsłużony wątek w jednym określonym wątku powodował zamknięcie interpretera, należy użyć metody egzemplarza o tej samej nazwie:

```
t = Thread.new { ... }
t.abort_on_exception = true
```

9.9.2. Wątki a zmienne

Jedną z kluczowych cech wątków jest ich zdolność do współdzielenia dostępu do zmiennych. Ponieważ wątki są definiowane przez bloki, mają one dostęp do wszystkich zmiennych (lokalnych, egzemplarza, globalnych itd.) znajdujących się w zasięgu tych bloków:

```
x = 0
t1 = Thread.new do
  # Ten wątek może sprawdzać i ustawiać wartość zmiennej x.
end
t2 = Thread.new do
  # Ten wątek również może sprawdzać i ustawiać wartość zmiennej x.
  # Może też sprawdzać i ustawiać zmienne t1 i t2.
end
```

Kiedy dwa lub więcej wątków równocześnie odczytuje i zapisuje te same zmienne, muszą uważać, aby robić to poprawnie. Więcej na ten temat piszemy w podrozdziale o synchronizacji.

9.9.2.1. Zmienne prywatne wątków

Zmienne zdefiniowane w bloku wątku są prywatne dla tego wątku, a więc nie są dostępne dla innych wątków. Jest to skutek reguł dotyczących zasięgu zmiennych w języku Ruby.

Często potrzebne jest, aby wątek posiadał prywatną kopię jakiejś zmiennej, by jego zachowanie nie zmieniło się, jeśli zmienna ta zostanie zmieniona. Spójrz na poniższy fragment kodu usiłującego utworzyć trzy wątki i wydrukować odpowiednio cyfry 1, 2 i 3:

```
n = 1
while n <= 3
  Thread.new { puts n }
  n += 1
end
```

W niektórych warunkach i niektórych implementacjach niniejszy kod działałby zgodnie z oczekiwaniemi i drukował cyfry 1, 2 i 3. W innych sytuacjach i innych implementacjach jednak mógłby tak nie zadziałać. Możliwe jest (jeśli nowo utworzone wątki nie zostają od razu uruchomione), że kod ten wydrukuje na przykład cyfry 4, 4 i 4. Każdy wątek odczytuje wspólnie zmienną `n`, a wartość tej zmiennej zmienia się w miarę wykonywania pętli. Wartość drukowana przez wątek zależy od tego, kiedy został on uruchomiony względem wątku nadziedzkiego.

Aby rozwiązać ten problem, przekazujesz bieżącą wartość `n` do metody `Thread.new` i przypisujesz bieżącą wartość tej zmiennej do parametru blokowego. Parametry blokowe są prywatne dla bloków (ale zobacz też podrozdział 5.4.3) i nie są współdzielone przez wątki:

```
n = 1
while n <= 3
  # Zapisanie prywatnej kopii bieżącej wartości zmiennej n w zmiennej x.
```

```
Thread.new(n) { |x| puts x }
n += 1
end
```

Należy zauważyć, że innym sposobem na rozwiązywanie tego problemu jest użycie iteratatora zamiast pętli `while`. W takim przypadku wartość zmiennej `n` jest prywatna dla zewnętrznego bloku i nigdy nie zmienia się podczas wykonywania tego bloku:

```
1.upto(3) { |n| Thread.new { puts n }}
```

9.9.2.2. Zmienne lokalne wątków

Niektóre specjalne zmienne globalne języka Ruby są **zmiennymi lokalnymi wątków** — mogą mieć różne wartości w różnych wątkach. Przykładami takich zmiennych są zmienna `$SAFE` (zobacz podrozdział 10.5) i zmienna `$~` (zobacz tabelę 9.3). Oznacza to, że jeśli dwa wątki równolegle przeprowadzają operację dopasowywania wyrażenia regularnego, każdy z nich będzie widział inną wartość dla zmiennej `$~`, a dopasowanie w jednym wątku nie będzie miało wpływu na wyniki dopasowania w drugim wątku.

Klasa `Thread` udostępnia coś w rodzaju tablicy asocacyjnej. Definiuje metody egzemplarza `[]` i `[]=` pozwalające na powiązanie dowolnych wartości z dowolnym symbolem (łańcuch zostanie przekonwertowany na symbol — w przeciwieństwie do prawdziwych tablic asocacyjnych, klasa `Thread` pozwala używać jako kluczy wyłącznie symboli). Wartości związane z tymi symbolami zachowują się jak zmienne lokalne wątków. Nie są prywatne jak zmienne lokalne bloków, ponieważ każdy wątek może sprawdzić wartość w dowolnym innym wątku. Nie są one jednak też zmiennymi współdzielonymi, ponieważ każdy wątek może posiadać ich własną kopię.

Jako przykład założymy, że utworzyłeś wątki do pobierania plików z serwera sieciowego. Główny wątek mógłby monitorować postęp tego pobierania. Aby to było możliwe, każdy wątek mógłby wykonywać następujące działania:

```
Thread.current[:progress] = bytes_received
```

Następnie ogólną liczbę pobranych bajtów główny wątek mógłby określić następująco:

```
total = 0
download_threads.each { |t| total += t[:progress] }
```

Poza metodami `[]` i `[]=` klasa `Thread` definiuje też metodę `key?` sprawdzającą, czy dla danego wątku istnieje określony klucz. Metoda `keys` zwraca tablicę symboli reprezentujących klucze zdefiniowane dla wątku. Niniejszy kod można lepiej napisać tak jak poniżej, aby działał dla wątków, które jeszcze nie zaczęły działać i nie zdefiniowały jeszcze klucza `:progress`:

```
total = 0
download_threads.each { |t| total += t[:progress] } if t.key?(:progress)}
```

9.9.3. Planowanie wykonywania wątków

Interpreter Ruby często ma do wykonania więcej wątków niż dostępnych procesorów. Kiedy nie jest możliwe prawdziwe przetwarzanie równolegle, jest ono symulowane poprzez dzielenie jednego procesora pomiędzy wątki. Proces dzielenia czasu procesora na kilka wątków nazywa się planowaniem wykonywania wątków. W zależności od implementacji i platformy planowanie wykonywania wątków może być wykonywane przez interpreter Ruby lub system operacyjny.

9.9.3.1. Priorytety wątków

Pierwszym czynnikiem mającym wpływ na planowanie wykonywania wątków są **priorytety wątków** — wykonywanie wątków z wysokim priorytetem jest planowane przed wątkami o niskim priorytecie. Mówiąc ściślej, wątek dostanie czas procesora tylko wtedy, gdy nie ma żadnych wątków z wyższym priorytetem od niego.

Do ustawiania i sprawdzania priorytetu obiektu klasy `Thread` służą metody `priority` i `priority=`. Zauważ, że nie da się ustawić priorytetu wątku przed jego uruchomieniem. Wątek może natomiast zwiększyć lub zmniejszyć swój priorytet jako pierwsza wykonana przez niego czynność.

Nowo utworzony wątek ma taki sam priorytet jak wątek, w którym został utworzony. Wątek główny ma początkowo priorytet 0.

Tak jak wiele innych aspektów wielowątkowości priorytety wątków zależą od implementacji języka Ruby i systemu operacyjnego, w którym on działa. Na przykład w systemie Linux wątki bez przywilejów nie mogą zmniejszać ani zwiększać swoich priorytetów. Dlatego w Ruby 1.9 (który używa wątków natywnych) ustawienia priorytetów wątków są przez system Linux ignorowane.

9.9.3.2. Wywłaszczanie wątków i metoda `Thread.pass`

Kiedy kilka wątków z takim samym priorytetem musi dzielić się jednym procesorem, decyzja, kiedy i jak długo będzie działał każdy z tych wątków, należy do algorytmu planującego. Niektóre algorytmy planujące wywłaszczają wątki, to znaczy pozwalają jednemu wątkowi działać tylko przez określoną ilość czasu, a następnie uruchamiają inny wątek o tym samym priorytecie. Inne algorytmy nie wywłaszczają wątków — uruchomiony wątek działa, dopóki nie zaśnie, nie zostanie zablokowany przez operację wejścia lub wyjścia albo nie obudzi się inny wątek z wyższym priorytetem.

Jeśli jakiś zachłanny wątek zajmujący dużo czasu (to znaczy taki, który nigdy nie blokuje się w celu wykonania operacji wejścia lub wyjścia) zostanie uruchomiony na niewywłaszczającym systemie planowania, „zagłodzi” on inne wątki z takim samym priorytetem, przez co nigdy nie będą one miały okazji działać. Aby tego uniknąć, wątki tego typu powinny co jakiś czas wywoływać metodę `Thread.pass` proszącą system planowania o przekazanie czasu procesora do innego wątku.

9.9.4. Stany wątków

Wątek może znajdować się w jednym z pięciu stanów. Dwa najbardziej interesujące z nich to stany żywych wątków — wątek, który jest żywy, jest **wykonywalny** lub **uśpiony**. Wątek, który jest wykonywalny, to taki, który aktualnie działa lub jest gotowy do działania, kiedy zwolnią się dla niego zasoby procesora. Wątek uśpiony to taki, który śpi (zobacz metodę `Kernel.sleep`), który oczekuje na operację wejścia lub wyjścia lub zatrzymał się (zobacz metodę `Thread.stop` opisaną poniżej). Wątki zazwyczaj przechodzą w tę i z powrotem pomiędzy stanami uśpienia i wykonywalności.

Dla wątków nieżywych są również dwa stany. Wątek, który zakończył działanie, mógł zakończyć je w sposób normalny lub nienormalny ze zgłoszeniem wyjątku.

Istnieje też jeden stan przejściowy. Wątek, który został zamknięty (zobacz metodę `Thread.kill` poniżej), ale jeszcze nie zakończył działania, jest w stanie **anulowania** (ang. *aborting*).

9.9.4.1. Sprawdzanie stanu wątków

Klasa `Thread` udostępnia kilka metod egzemplarza pozwalających sprawdzić stan wątku. Metoda `alive?` zwraca wartość `true`, jeśli wątek jest wykonywalny lub uśpiony. Metoda `stop?` zwraca wartość `true`, jeśli wątek jest w innym stanie niż wykonywalny. W końcu metoda `status` zwraca stan wątku. Istnieje pięć wartości zwrotnych odpowiadających pięciu stanom wątków. Zestawienie tych wartości przedstawia poniższa tabela:

Stan wątku	Wartość zwrotna
Wykonywalny	<code>run</code>
Uśpiony	<code>sleep</code>
Anulowany	<code>aborting</code>
Zakończony normalnie	<code>false</code>
Zakończony przez wyjątek	<code>nil</code>

9.9.4.2. Zmienianie stanu — wstrzymywanie, budzenie i zamykanie wątków

Wątki są tworzone w stanie **uruchamialnym** i mogą od razu działać. Wątek może zostać wstrzymany — wejść w stan **uśpienia** — poprzez wywołanie metody `Thread.sleep`. Jest to metoda klasowa operująca na bieżącym wątku — nie ma odpowiadającej jej metody egzemplarza, a więc jeden wątek nie może zmusić innego do wstrzymania. Wywołanie metody `Thread.stop` wywołuje taki sam efekt jak wywołanie metody `Kernel.sleep` bez argumentów — wątek wstrzymuje działanie na zawsze (lub aż zostanie obudzony, o czym mowa poniżej).

Wątki wchodzą też w okresowy stan **uśpienia**, jeśli wywołają metodę `Kernel.sleep` z argumentem. W takim przypadku budzą się automatycznie i powracają do wykonywalnego stanu po (w przybliżeniu) upływie określonej liczby sekund. Wywołanie blokującej metody klasy `IO` również może spowodować przejście wątku w stan uśpienia, dopóki nie zakończy się operacja wejścia lub wyjścia — w rzeczywistości to właśnie dzięki opóźnieniu związanemu z operacjami wejścia i wyjścia wielowątkowość ma sens w systemach jednoprocesorowych.

Wątek wstrzymany za pomocą metody `Thread.stop` lub `Thread.sleep` można uruchomić ponownie (nawet jeśli nie upłynął jeszcze wyznaczony czas uśpienia) za pomocą metod egzemplarza `wakeup` i `run`. Obie te metody przełączają stan wątku z **uśpionego** na **wykonywalny**. Metoda `run` wywołuje też system planowania. Powoduje to, że bieżący wątek zwraca zajmowane zasoby procesora i może spowodować natychmiastowe uruchomienie nowo obudzonego wątku. Metoda `wakeup` budzi wątek, ale nie zmusza innego wątku do zwrócenia zasobów procesora.

Wątek może zmienić swój stan z **wykonywalnego** na jeden z **zakończonych** poprzez wyjście ze swojego bloku lub zgłoszenie wyjątku. Innym sposobem na normalne zakończenie wątku jest wywołanie metody `Thread.exit`. Pamiętaj, że przed zakończeniem działania wątku w ten sposób wykonywane są wszystkie klauzule `ensure`.

Jeden wątek może zmusić inny wątek do zamknięcia, wywołując na jego rzecz metodę `kill` tego wątku. Metody `terminate` i `exit` są synonimami tej metody. Metody te ustawiają wątek w stanie **zakończony normalnie**. Aby zgłosić w innym wątku wyjątek, należy wywołać metodę

egzemplarza `raise`. Jeśli dany wątek nie może obsłużyć zgłoszonego w nim wyjątku, wejdzie w stan **zakończony przez wyjątek**. Klauzule `ensure` wątków są wykonywane w taki sam sposób jak podczas normalnego procesu propagacji wyjątku.

Zamykanie wątku jest niebezpieczne, jeśli nie wiadomo, czy nie jest on w trakcie zmieniania współdzielonego stanu systemu. Zamknięcie wątku za pomocą jednej z metod zakończonych znakiem ! jest jeszcze bardziej niebezpieczne, ponieważ tak zamknięty wątek może pozostać otwarte pliki, gniazda lub inne zasoby. Jeśli wątek musi mieć możliwość zamknięcia na polecenie, lepiej jest sprawdzać w nim co jakiś czas zmienną zawierającą odpowiedni znaczek i elegancko kończyć jego działanie, kiedy zmienna ta zostanie odpowiednio ustawiona.

9.9.5. Tworzenie list i grup wątków

Metoda `Thread.list` zwraca tablicę obiektów klasy `Thread` reprezentujących wszystkie żywe (działające lub uśpione) wątki. Kiedy wątek jest zamknięty, zostaje usunięty z tej tablicy.

Każdy wątek poza głównym jest tworzony przez jakiś inny wątek. W związku z tym wątki można zorganizować w strukturę drzewa, w której każdy wątek ma rodzica i potomków. Jednak klasa `Thread` nie udostępnia tych informacji — wątki są uważane za byty autonomiczne, a nie podporządkowane wątkom, które je utworzyły.

Aby narzuścić porządek na podzbior wątków, można utworzyć obiekt klasy `ThreadGroup` i dodać do niego wątki:

```
group = ThreadGroup.new
3.times { |n| group.add(Thread.new { do_task(n) }) }
```

Nowe wątki są początkowo dodawane do grupy, do której należy ich rodzic. Do wysyłania zapytań do obiektu klasy `ThreadGroup`, do którego należy wątek, służy metoda egzemplarza `group`. Aby utworzyć tablicę wątków znajdujących się w grupie, należy użyć metody `list` klasy `ThreadGroup`. Podobnie jak klasowa metoda `Thread.list`, metoda egzemplarza `ThreadGroup`.
→`list` zwraca tylko te wątki, które jeszcze nie zostały zamknięte. Przy użyciu tej metody `list` można zdefiniować metody działające na wszystkich wątkach w grupie. Metoda taka mogłaby na przykład zmniejszać priorytet wszystkich wątków w wybranej grupie.

To, co sprawia, że klasa `ThreadGroup` jest bardziej użyteczna od prostej tablicy wątków, jest jej metoda `enclose`. Po wywołaniu tej metody na rzecz grupy wątków należące do niej wątki nie mogą zostać z niej usunięte i nie można dodawać nowych. Wątki w tej grupie mogą tworzyć nowe wątki, które także stają się członkami tej grupy. Tego typu grupy są przydatne przy wykonywaniu niezaufanego kodu Ruby pod zmienną `$SAFE` (zobacz podrozdział 10.5), kiedy programista chce wiedzieć, ile wątków utworzył ten kod.

9.9.6. Przykłady programów wielowątkowych

Znając już model wątków w języku Ruby i API, przyjrzyj się kilku przykładom prawdziwie wielowątkowego kodu.

9.9.6.1. Współbieżny odczyt plików

Wątki są najczęściej wykorzystywane w programach ograniczonych operacjami wejścia i wyjścia. Dzięki nim program może wykonywać różne działania, oczekując na przyjście danych od użytkownika, z systemu plików czy sieci. Na przykład w poniższym kodzie zdefiniowano

metodę `conread` pobierającą tablicę nazw plików i zwracającą tablicę asocjacyjną rzutującą te nazwy na zawartość tych plików. Do równoległego odczytu tych plików wykorzystywane są wątki. Program ten jest przeznaczony do użytku z modelem `open-uri`, który pozwala na otwieranie adresów HTTP i FTP za pomocą metody `Kernel.open` i czytać je tak, jakby były plikami:

```
# Równolegle wczytuje pliki. Należy używać z modelem open-uri ładującym adresy URL.
# Przekazuje tablicę nazw plików. Zwraca tablicę asocjacyjną rzutującą te nazwy plików na ich treść.
def conread(filenames)
  h = {}                                # Pusta tablica asocjacyjna wyników.
  # Utworzenie po jednym wątku dla każdego pliku.
  filenames.each do |filename|          # Dla każdego wyznaczonego pliku.
    h[filename] = Thread.new do           # Utworzenie wątku, rzutowanie na nazwę pliku.
      open(filename) {|f| f.read }        # Otwarcie i odczyt pliku.
    end                                  # Wartość wątku jest treścią pliku.
  end
  # Iteracja przez tablicę asocjacyjną z oczekaniem, aż każdy wątek skończy działanie.
  # Zastąpienie wątku w tablicy asocjacyjnej jego wartością (treścią pliku).
  h.each_pair do |filename, thread|
    begin
      h[filename] = thread.value       # Rzutowanie nazwy pliku na treść pliku.
    rescue
      h[filename] = $!                # lub na zgłoszony wyjątek.
    end
  end
end
```

9.9.6.2. Serwer wielowątkowy

Innym, prawie klasycznym, zastosowaniem wątków jest pisanie serwerów, które pozwalają na podłączenie się wielu klientów jednocześnie. Widziałeś, jak to zrobić przy użyciu zwielokrotniania za pomocą metody `Kernel.select`, ale istnieje też prostsze rozwiązanie (chociaż może mniej skalonalne) polegające na wykorzystaniu wątków:

```
require 'socket'
# Niniejsza metoda wymaga gniazda połączonego z klientem.
# Wczytuje dane od klienta, odwraca je i wysyla z powrotem.
# Metoda ta może być uruchomiona przez wiele wątków jednocześnie.
def handle_client(c)
  while true
    input = c.gets.chomp          # Odczytanie wiersza danych od klienta.
    break if !input               # Wyjście, jeśli nie ma więcej danych
    break if input=="quit"        # lub jeśli klient tego żąda.
    c.puts(input.reverse)         # W przeciwnym przypadku wysłanie odpowiedzi do klienta.
    c.flush                      # Wymuszenie wyjścia danych.
  end
  c.close                        # Zamknięcie gniazda klienta.
end
server = TCPServer.open(2000) # Nasłuchiwanie na porcie 2000.
while true                     # Serwery działają w nieskończoność.
  client = server.accept       # Poczekanie, aż klient się połączy.
  Thread.start(client) do |c|   # Uruchomienie nowego wątku.
    handle_client(c)            # Obsługa klienta w tym wątku.
  end
end
```

9.9.6.3. Iteratory współbieżne

Mimo że zadania związane z wejściem i wyjściem są typowym przypadkiem użycia wątków w języku Ruby, nie są one ograniczone tylko do tego zastosowania. Poniższy kod dodaje do modułu `Enumerable` metodę `conmap`. Działa ona podobnie do metody `map`, ale każdy element tablicy wejściowej przetwarza w osobnym wątku:

```

module Enumerable
  def conmap(&block)
    threads = []
    self.each do |item|
      # Wywołuje blok w nowym wątku i zapamiętuje ten wątek.
      threads << Thread.new { block.call(item) }
    end
    # Rzutowanie tablicy wątków na ich wartości.
    threads.map { |t| t.value } # Zwrócenie tablicy wartości.
  end
end

```

Poniżej znajduje się podobna współbieżna wersja iteratora each:

```

module Enumerable
  def concurrently
    map { |item| Thread.new { yield item } }.each { |t| t.join }
  end
end

```

Kod ten jest zwięzły, a zarazem wymagający — jeśli rozumiesz go, jesteś na dobrej drodze do opanowania składni i iteratorów Ruby.

Przypomnijmy, że w Ruby 1.9 iteratory, którym nie zostanie przekazany blok, zwracają obiekt enumeratora. Oznacza to, że mając zdefiniowaną wcześniej metodę concurrently i obiekt klasy Hash o nazwie h, można napisać:

```
h.each_pair.concurrently { |*pair| process(pair) }
```

9.9.7. Wątki a zakleszczenia

Jeśli przynajmniej dwa wątki współdzielą dostęp do tych samych danych i jeden z nich modyfikuje te dane, programista musi zapewnić, aby żaden inny wątek nie mógł uzyskać dostępu do tych danych w trakcie przetwarzania ich przez inny wątek. Nazywa się to **wykluczaniem wątków**. Po przedstawieniu przykładów stanie się jasne, dlaczego jest to niezbędne.

Najpierw wyobraź sobie, że dwa wątki przetwarzają pliki i każdy z nich inkrementuje wartość współdzielonej zmiennej, aby wiedzieć, ile w sumie plików zostało przetworzonych. Problem polega na tym, że inkrementacja wartości zmiennej nie jest operacją niepodzielną, to znaczy nie odbywa się w jednym etapie. Aby zwiększyć wartość zmiennej, program musi odczytać jej wartość, dodać 1, a następnie zapisać nową wartość z powrotem w tej zmiennej. Założymy, że licznik wskazuje aktualnie wartość 100. Przeanalizuj zatem następujący proces przeplataneego wykonywania wątków — pierwszy wątek odczytuje wartość 100, ale zanim doda 1, system planujący zatrzymuje go i pozwala działać drugiemu wątkowi. Teraz drugi wątek odczytuje wartość 100, dodaje do niej 1 i zapisuje w zmiennej wartość 101. Ten sam wątek zaczyna w tym momencie odczyt nowego pliku, co powoduje jego zablokowanie i zezwolenie pierwszemu wątkowi na wznowienie działania. Wątek ten zwiększa 100 o 1 i zapisuje wynik. Oba wątki zwiększyły licznik, ale jego wartość wynosi 101, a nie 102.

Innym klasycznym przykładem sytuacji, w której konieczne jest wykluczanie wątków, to system bankowości elektronicznej. Założymy, że jeden wątek przelewa pieniądze z rachunku oszczędnościowego na rachunek bieżący, a inny wątek generuje miesięczne raporty do wysłania klientom. Bez odpowiedniego wykluczania wątków wątek generujący raporty mógłby odczytać dane kont klientów po pobraniu środków z rachunku oszczędnościowego, a przed uznaniem ich na rachunku bieżącym.

Tego typu problemy rozwiązywane są za pomocą kooperacyjnego mechanizmu blokującego. Każdy wątek chcący uzyskać dostęp do danych musi je najpierw **zablokować**. Blokadę reprezentuje obiekt klasy Mutex (nazwa pochodzi od angielskich słów *mutual exclusion*, czyli wzajemne wykluczanie). Aby zablokować obiekt klasy Mutex, należy na jego rzecz wywołać metodę `lock`. Po zakończeniu odczytu lub modyfikacji współdzielonych danych należy wywołać metodę `unlock` tego obiektu klasy Mutex. Metoda `lock` blokuje, kiedy zostaje wywołana na rzecz obiektu klasy Mutex, który jest już zablokowany, i nie zwraca wartości, zanim algorytm wywołujący nie uzyska blokady. Jeśli każdy wątek mający dostęp do współdzielonych danych prawidłowo blokuje i odblokowuje obiekt klasy Mutex, żaden wątek nie zobaczy danych w niespójnym stanie, dzięki czemu nie wystąpią opisane wyżej problemy.

Klasa Mutex wchodzi w skład rdzenia języka Ruby 1.9, a w Ruby 1.8 jest częścią standardowej biblioteki `thread`. Zamiast metod `lock` i `unlock` częściej używana jest metoda `synchronize` z blokiem. Metoda `synchronize` blokuje obiekt klasy Mutex, wykonuje kod w bloku i odblokowuje obiekt klasy Mutex w klauzuli `ensure`, dzięki czemu wyjątki są obsługiwane prawidłowo. Poniżej znajduje się prosty model opisanego wcześniej konta bankowego. Dostęp do współdzielonych danych jest synchronizowany przez obiekt klasy Mutex:

```
require 'thread' # Aby udostępnić klasę Mutex w Ruby 1.8.  
# Konto bankowe ma nazwę (name), określoną ilość środków bieżących (checking) i określoną kwotę oszczędności (savings).  
class BankAccount  
  def init(name, checking, savings)  
    @name, @checking, @savings = name, checking, savings  
    @lock = Mutex.new # Dla bezpieczeństwa wątków.  
  end  
  # Blokuje konto i przelewa środki z rachunku oszczędnościowego na bieżący.  
  def transfer_from_savings(x)  
    @lock.synchronize {  
      @savings -= x  
      @checking += x  
    }  
  end  
  # Blokuje konto i generuje raport o saldach bieżących.  
  def report  
    @lock.synchronize {  
      "#@name\nChecking: #@checking\nSavings: #@savings"  
    }  
  end  
end
```

9.9.7.1. Zakleszczenie

Używając obiektów klasy Mutex do wykluczania wątków, trzeba uważać na **zakleszczenie**. Zakleszczenie to sytuacja, kiedy wszystkie wątki czekają na dostęp do danych będących w posiadaniu innego wątku. Ponieważ wszystkie wątki są zablokowane, nie mogą zwolnić swoich blokad. Ponieważ nie mogą one zwolnić swoich blokad, żaden inny wątek nie może ich założyć.

Klasyczne zakleszczenie ma miejsce przy udziale dwóch wątków i dwóch obiektów klasy Mutex. Wątek 1. blokuje Mutex 1., a następnie próbuje zablokować Mutex 2. W międzyczasie wątek 2. blokuje Mutex 2., a następnie próbuje zablokować Mutex 1. Żaden z tych wątków nie może założyć potrzebnej blokady i żaden nie może zwolnić blokady, której potrzebuje ten drugi. W związku z tym wątki zostają zablokowane na zawsze:

```
# Klasyczne zakleszczenie — dwa wątki i dwie blokady.  
require 'thread'  
m, n = Mutex.new, Mutex.new  
t = Thread.new {
```

```

m.lock
puts "Wątek t zablokował Mutex m."
sleep 1
puts "Wątek t czeka na zablokowanie obiektu Mutex n."
n.lock
}
s = Thread.new {
n.lock
puts "Wątek s zablokował Mutex n."
sleep 1
puts "Wątek s czeka na zablokowanie obiektu Mutex m."
m.lock
}
t.join
s.join

```

Sposobem na uniknięcie tego typu zakleszczenia jest blokowanie zasobów zawsze w tej samej kolejności. Jeśli drugi wątek zablokowałby obiekt `m` przed obiektem `n`, zakleszczenie by nie powstało.

Pamiętaj, że zakleszczenie może się zdarzyć także bez użycia obiektów klasy `Mutex`. Wywołanie metody `join` na rzecz wątku wywołującego metodę `Thread.stop` również zakleszczy te dwa wątki, chyba że jest trzeci wątek mogący obudzić ten zatrzymany wątek.

Warto też wiedzieć, że niektóre implementacje języka Ruby mogą wykrywać tego typu proste zakleszczenia i zamykać program z błędem, ale nie jest to gwarantowane.

9.9.8. Klasy Queue i SizedQueue

Standardowa biblioteka `thread` udostępnia struktury danych `Queue` i `SizedQueue` przeznaczone specjalnie do użytku w programowaniu współbieżnym. Implementują one bezpieczne dla wątków kolejki typu FIFO i są przeznaczone dla modelu programowania producent-konsument. W tym modelu jeden wątek produkuje dane i umieszcza je w kolejce za pomocą metody `enq` lub jej synonimu `push`. Inny wątek natomiast „konsumuje” te dane, usuwając je z kolejki za pomocą metody `deq` (synonimami metody `deq` są metody `pop` i `shift`).

Kluczową cechą klasy `Queue`, która sprawia, że jest ona przydatna w programowaniu współbieżnym, jest to, że metoda `deq` blokuje, jeśli kolejka jest pusta, i czeka, aż wątek producent wstawi do niej jakieś dane. Klasy `Queue` i `SizedQueue` implementują to samo podstawowe API, z tym, że klasa `SizedQueue` ma określony maksymalny rozmiar. Jeśli kolejka osiągnie swój maksymalny rozmiar, metoda wstawiająca do niej dane zablokuje ją, aż wątek konsument usunie z niej jakieś dane.

Tak samo jak w przypadku innych klas kolekcyjnych, liczbę elementów w kolejce można sprawdzić za pomocą metod `size` i `length`, a do sprawdzania, czy kolejka jest pusta, służy metoda `empty?`. Maksymalny rozmiar kolejki `SizedQueue` należy określić przy wywołaniu metody `SizedQueue.new`. Po utworzeniu kolejki `SizedQueue` jej maksymalny rozmiar można sprawdzać i zmodyfikować za pomocą metod `max` i `max=`.

We wcześniejszej części niniejszego rozdziału poznaleś sposób dodania do modułu `Enumerable` współbieżnej metody `map` (`conmap`). Teraz zdefiniujemy metodę łączącą współbieżną metodę `map` ze współbieżną metodą `inject`. Będzie ona tworzyła wątek dla każdego elementu kolekcji umożliwiającej iterację i wykorzystywała ten wątek do zastosowania rzutującego obiektu

klasy Proc. Wartość zwrócona przez ten obiekt klasy Proc jest ustawiana w kolejce do obiektu klasy Queue. Jeden ostatni wątek działa jak konsument — usuwa dane z kolejki i przekazuje je do wstawiającego obiektu klasy Proc, kiedy stają się dostępne.

Tę wspólniezną metodę wstawiającą nazwiemy `conject`. Można jej użyć do wspólnieznego obliczania sumy kwadratów wartości znajdujących się w tablicy. Zauważ jednak, że sekwencyjny algorytm prawie na pewno byłby szybszy w tego typu prostych operacjach obliczania sum kwadratów:

```
a = [-2,-1,0,1,2]
mapper = lambda { |x| x*x }           # Obliczanie kwadratów.
injector = lambda { |total,x| total+x } # Obliczanie sumy.
a.conject(0, mapper, injector)        # => 10.
```

Kod źródłowy metody `conject` jest następujący — zwróć uwagę na użycie obiektu klasy Queue i jego metod `enq` i `deq`:

```
module Enumerable
  # Współbieżna metoda inject — przyjmuje wartość początkową i dwa obiekty klasy Proc.
  def conject(initial, mapper, injector)
    # Użycie kolejki do przekazania wartości z wątków rzutujących do wątku wstawiającego.
    q = Queue.new
    count = 0                         # Ile elementów?
    each do |item|
      Thread.new do
        q.enq(mapper[item])          # Dla każdego elementu
      end                            # tworzy nowy wątek.
      count += 1                      # Liczanie elementów.
    end
    t = Thread.new do                  # Utworzenie wątku wstawiającego.
      x = initial                   # Start z określona wartością początkową.
      while(count > 0)              # Jedno powtórzenie pętli dla każdego elementu.
        x = injector[x, q.deq]       # Usunięcie wartości z kolejki i wstawienie jej.
        count -= 1                  # Odliczanie w dół.
      end
      x                           # Wartość wątku to wstawiona wartość.
    end
    t.value   # Poczekanie na wątek wstawiający i zwrócenie jego wartości.
  end
end
```

9.9.9. Zmienne warunku i kolejki

W klasie Queue należy zauważać jedną ważną rzecz — metoda `deq` może blokować. Normalnie blokowanie uznaje się za operację związaną z metodami klasy IO (lub z wywoływaniem metody `join` na rzecz wątku albo metody `lock` na rzecz obiektu klasy Mutex). W programowaniu współbieżnym czasami jednak konieczne jest zmuszenie wątku do poczekania, aż jakiś warunek (znajdujący się poza jego kontrolą) stanie się prawdziwy. W przypadku klasy Queue warunkiem tym jest niepusta kolejka — jeśli kolejka jest pusta, konsument musi poczekać, aż producent wywoła metodę `enq` wstawiającą dane do kolejki.

Aby zmusić wątek do czekania, aż jakiś inny wątek poinformuje go, że może kontynuować, najlepiej użyć klasy `ConditionVariable`. Podobnie jak klasa `Queue`, klasa `ConditionVariable` jest częścią standardowej biblioteki `thread`. Obiekty klasy `ConditionVariable` tworzy się za pomocą metody `ConditionVariable.new`. Aby zmusić wątek do poczekania na spełnienie warunku, należy użyć metody `wait`. Aby obudzić oczekującą wątek, należy użyć metody `signal`. Aby obudzić wszystkie oczekujące wątki, należy wywołać metodę `broadcast`. Z używaniem

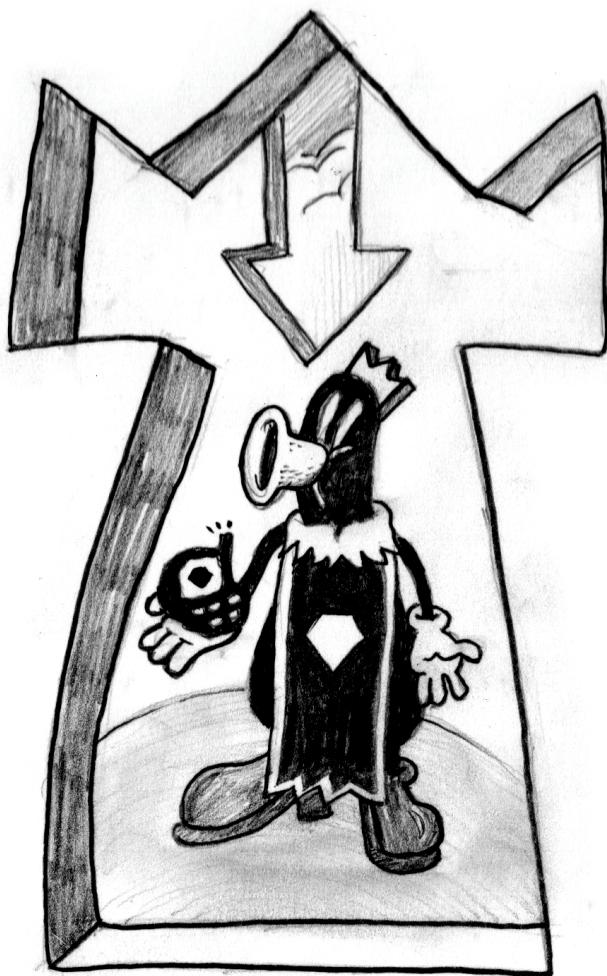
zmiennych warunku wiąże się tylko jeden haczyk — aby wszystko działało jak należy, oczekujący wątek musi przekazać zablokowany obiekt klasy Mutex do metody wait. Obiekt ten zostanie tymczasowo odblokowany podczas oczekiwania przez wątek i zostanie z powrotem zablokowany, kiedy wątek się obudzi.

Dyskusję na temat współbieżności zamkamy klasą użytkową, która czasami przydaje się we współbieżnych programach. Nazywa się Exchanger, a umożliwia dwóm wątkom wymianę dowolnych wartości. Założymy, że mamy dwa wątki t1 i t2 i obiekt klasy Exchanger o nazwie e. Wątek t1 wywołuje metodę e.exchange(1). Metoda ta blokuje (za pomocą obiektu klasy ConditionVariable), aż wątek t2 wywoła metodę e.exchange(2). Ten drugi wątek nie blokuje, tylko zwraca wartość 1 — wartość przekazaną przez wątek t1. Teraz, kiedy drugi wątek wywołał metodę exchange, wątek t1 budzi się ponownie i zwraca wartość 2 z metody exchange.

Przedstawiona tu implementacja klasy Exchanger jest dosyć skomplikowana, ale demonstruje typowe użycie klasy ConditionVariable. Interesującą cechą tego kodu jest użycie w nim dwóch obiektów klasy Mutex. Jeden z nich jest używany do synchronizacji dostępu do metody exchange. Jest on przekazywany do metody wait zmiennej warunku. Drugi obiekt klasy Mutex pozwala określić, czy wątek wywołujący wywołał metodę exchange jako pierwszy czy drugi. Zamiast metody lock z tym obiektem klasy Mutex niniejsza klasa używa nieblokującej metody try_lock. Jeśli wywołanie @first.try_lock zwraca wartość true, wywołujący wątek jest pierwszy. W przeciwnym przypadku jest drugi:

```
require 'thread'
class Exchanger
  def initialize
    # Zmienne do przechowywania wartości do wymiany.
    @first_value = @second_value = nil
    # Obiekt klasy Mutex chroniący dostępu do metody exchange.
    @lock = Mutex.new
    # Obiekt klasy Mutex pozwalający sprawdzić, czy metoda exchange
    # jest wywoływana po raz pierwszy, czy drugi.
    @first = Mutex.new
    # Obiekt klasy ConditionVariable pozwalający pierwszemu wątkowi poczekać
    # na przybycie drugiego wątku.
    @second = ConditionVariable.new
  end
  # Zamienia tę wartość na wartość przekazaną przez drugi wątek.
  def exchange(value)
    @lock.synchronize do
      if @first.try_lock
        # Tylko jeden wątek może wywołać tę metodę w danym momencie.
        # To jest pierwszy wątek.
        @first_value = value # Zapisanie argumentu pierwszego wątku.
        # Poczekanie na przybycie drugiego wątku.
        # Tymczasowo odblokowuje obiekt klasy Mutex podczas oczekiwania,
        # dzięki czemu drugi wątek też może wywołać tę metodę.
        @second.wait(@lock) # Poczekanie na drugi wątek.
        @first.unlock # Przygotowanie do kolejnej wymiany.
        @second_value = value # Zwrócenie wartości drugiego wątku.
      else
        # W przeciwnym przypadku ten wątek jest drugi.
        @second_value = value # Zapisanie drugiej wartości.
        @second.signal # Poinformowanie drugiego wątku, że jesteś tutaj.
        @first_value = value # Zwrócenie wartości pierwszego wątku.
      end
    end
  end
end
```


Środowisko Ruby



Niniejszy rozdział opisuje wszystkie tematy programistyczne, których nie poruszono jeszcze do tej pory. Większość z opisywanych tu własności ma związek z interfejsem między językiem Ruby a systemem operacyjnym, w którym on działa. Dlatego niektóre z tych cech są zależne od konkretnego systemu operacyjnego. Tak samo niektóre z nich mogą być zależne od konkretnej implementacji — nie każdy interpreter Ruby implementuje je w taki sam sposób. Tematy opisane w tym rozdziale to:

- Argumenty wiersza poleceń i zmienne środowiskowe.
- Środowisko wykonawcze najwyższego poziomu — funkcje globalne, zmienne i stałe.
- Skróty dla skryptów przetwarzających tekst — funkcje globalne, zmienne i opcje interpretera w większości zainspirowane przez język Perl, które pozwalają na pisanie krótkich, ale mających duże możliwości programów do przetwarzania plików tekstowych.
- Polecenia systemu operacyjnego — wykonywanie poleceń powłoki i wywoływanie plików wykonywalnych w systemie operacyjnym. Dzięki tym cechom język Ruby może być używany jako język skryptowy.
- Bezpieczeństwo — sposoby zmniejszania ryzyka ataków typu SQL injection i innych podobnych ataków przy użyciu mechanizmu obiektów niepewnych oraz umieszczania niezaufanego kodu w ograniczonym środowisku wykonawczym (ang. *sandbox*) z poziomem zabezpieczeń \$SAFE.

10.1. Uruchamianie interpretera Ruby

Standardową implementację opartą na języku C uruchamia się w wierszu poleceń w następujący sposób:

```
ruby [opcje] [--] program [argumenty]
```

W miejscu opcje można wstawić zero lub więcej argumentów wiersza poleceń, które zmieniają działanie interpretera. Dostępne argumenty są opisane nieco dalej.

W miejscu program należy wpisać nazwę pliku z programem Ruby, który ma zostać uruchomiony. Jeśli nazwa programu zaczyna się od łącznika, należy przed nią postawić znaki --, aby nie została potraktowana jako opcja. W przypadku gdy nazwę programu stanowi sam łącznik lub zostanie opuszczona część program albo argumenty, interpreter wczyta tekst programu ze standardowego strumienia wejściowego.

Ostatni człon argumenty to dowolna liczba dodatkowych tokenów w wierszu poleceń. Tokeny te stają się elementami tablicy ARGV.

W kolejnych podrozdziałach opisane są opcje obsługiwane przez standardową implementację języka Ruby opartą na języku C. Warto zauważyć, że aby użyć którejś z opcji -W, -w, -v, -d, -I, -r, -K, -E i -T, można ustawić zmienną środowiskową RUBYOPT. Opcje będą automatycznie włączane przy każdym uruchomieniu interpretera, tak jakby zostały podane w wierszu poleceń, chyba że użyto opcji --disable-rubyopt.

10.1.1. Najczęściej używane opcje

Przedstawione poniżej opcje są prawdopodobnie najczęściej używane. Można się spodziewać, że większość implementacji języka Ruby je obsługuje lub dostarcza dla nich działające alternatywy:

-w

Włącza ostrzeżenia o wycofywanym lub problematycznym kodzie i ustawia zmienną \$VERBOSE na wartość true. Wielu programistów języka Ruby używa tej zmiennej rutynowo, aby mieć pewność, że pisany przez nich kod jest całkowicie czysty.

-e skrypt

Wykonuje kod Ruby zapisany w miejscu skrypt. Jeśli jest więcej niż jedna opcja -e, związane z nimi skrypty są traktowane jako osobne wiersze kodu. Ponadto interpreter w takiej sytuacji nie ładuje ani nie uruchamia żadnego programu wyznaczonego w wierszu poleceń.

Aby umożliwić wykonywanie jednowierszowych skryptów, kod Ruby podany za pomocą opcji -e może używać skrótu dopasowującego Regexp, który został opisany dalej w tym rozdziale.

-I ścieżka

Dodaje katalogi ścieżka na początku globalnej tablicy \$LOAD_PATH. Tablica ta określa, które katalogi mają zostać przeszukane przez metody load i require (ale nie zmienia sposobu ładowania programu wyznaczonego w wierszu poleceń).

W wierszu poleceń można użyć kilku opcji -I i każda z nich może określać jeden lub więcej katalogów. Jeśli jedna opcja -I zawiera listę kilku katalogów, powinny one być oddzielone znakami : w systemach typu Unix i ; w systemach Windows.

-r biblioteka

Ładuje wyznaczoną bibliotekę przed uruchomieniem wyznaczonego programu. Opcja ta działa tak, jakby pierwszy wiersz programu był następujący:

```
require 'library'
```

Spacja pomiędzy opcją -r a nazwą biblioteki jest opcjonalna i często się ją opuszcza.

-rubygems

Nie jest to prawdziwa opcja, a sprytny sposób użycia opcji -r. Ładuje moduł o nazwieubygems (bez r) z biblioteki standardowej. Moduł ubygems dla wygody ładuje tylko prawdziwy moduł rubygems. W Ruby 1.9 zainstalowane gemy można ładować bez tego modułu, a więc opcja ta jest potrzebna tylko w Ruby 1.8.

--disable-gems

Opcja Ruby 1.9. Uniemożliwia dodawanie katalogów instalacyjnych gemów do domyślnej ścieżki wczytywania. Jeśli w systemie jest zainstalowana duża liczba gemów, które nie są używane w aktualnie uruchamianym programie (lub uruchamiany program bezpośrednio zarządza swoimi zależnościami za pomocą metody gem), opcja ta może przyspieszyć uruchamianie takiego programu.

-d, --debug

Ustawiają zmienne globalne \$DEBUG i \$VERBOSE na wartość true. Kiedy zmienne te są ustawione, program lub używana przez niego biblioteka może drukować dane debugera lub wykonywać inne działania.

-h

Wyświetla listę opcji interpretera i kończy działanie.

10.1.2. Opcje ostrzeżeń i informacji

Poniższe opcje ustawiają rodzaj lub ilość informacji wyświetlanych przez interpreter Ruby:

-W, -W2, --verbose

Wszystkie te opcje są synonimami opcji -w — włączają pełne ostrzeżenia i ustawiają zmienną \$VERBOSE na wartość true.

-W0

Tłumi wszystkie ostrzeżenia.

-v

Drukuję numer wersji języka Ruby. Jeśli nie zostanie wyznaczony żaden program, kończy działanie, nie wczytując programu ze standardowego wejścia. W przypadku gdy program jest wyznaczony, uruchamia go tak, jakby opcja --verbose (lub -w) była włączona.

--version, --copyright, --help

Drukują numer wersji języka Ruby, informacje o prawach autorskich lub pomoc wiersza poleceń. Opcja --help jest synonimem opcji -h. Opcja --version różni się od opcji -v tym, że nigdy nie uruchamia wyznaczonego programu.

10.1.3. Opcje kodowania

Poniższe opcje służą do określania domyślnego kodowania zewnętrznego procesu Ruby oraz domyślnego kodowania źródła dla plików, które nie określają własnego kodowania w komentarzu kodowania. Jeśli żadna z tych opcji nie zostanie podana, domyślne zewnętrzne kodowanie jest określone na podstawie lokalizacji, a domyślne kodowanie źródła to ASCII (więcej informacji na temat kodowania źródła i kodowania zewnętrznego znajduje się w podrozdziale 2.4):

-Kkod

W Ruby 1.8 opcja ta określa kodowanie źródła skryptu i ustawia zmienną globalną \$KCODE.

W Ruby 1.9 ustawia domyślne kodowanie zewnętrzne procesu Ruby i określa domyślne kodowanie źródła.

Kody a, A, n i N odpowiadają ASCII. Kody u i U to Unicode, e i E — EUC-JP, a s i S — SJIS (EUC-JP i SJIS to popularne kodowania japońskie).

-E kodowanie, --encoding=kodowanie

Opcje te działają podobnie do opcji -K, ale pozwalają na określenie kodowania za pomocą jego nazwy zamiast jednoliterowego skrótu.

10.1.4. Opcje przetwarzania tekstu

Poniższe opcje modyfikują domyślny sposób przetwarzania tekstu lub są przydatne przy pisaniu jednowierszowych skryptów wykonywanych za pomocą opcji -e:

-0xxx

Opcja ta jest cyfrą zero, nie literą O. W miejscie xxx należy wstawić od zera do trzech cyfr ósemkowych. Cyfry te są kodem ASCII znaku oddzielającego rekordy danych wejściowych i ustawiają zmienną \$/. W ten sposób definiowane są wiersze dla metody gets i innych podobnych metod. Sama opcja -0 ustawia zmienną \$/ na kod 0. Opcja -00 jest wyjątkowa. Przestawia Ruby na „tryb akapitowy”, w którym wiersze są oddzielane przez dwa kolejne znaki nowego wiersza.

-a

Automatycznie rozbija każdy wiersz danych wejściowych na pola i zapisuje je w zmiennej \$F. Działa tylko z opcjami pętlowymi -n i -p oraz dodaje kod \$F = \$_.split na początek każdej iteracji. Zobacz też opcję -F.

-F wzorzec

Ustawia separator pól na wejściu (\$;). Ma to wpływ na działanie metody split wywoływanej bez argumentów. Zobacz opcję -a.

W miejscu wzorzec można wstawić pojedynczy znak lub dowolne wyrażenie regularne bez ukośników. W zależności od używanej powłoki może być konieczne umieszczenie ukośników w cudzysłowach lub podwojenie ich w wyrażeniach regularnych wpisywanych w wierszu polecień.

-i [rozszerzenie]

Edytuje pliki wyznaczone w wierszu polecień na miejscu. Wczytuje dane z tych plików, a dane wyjściowe z powrotem zapisuje w tych plikach. Jeśli zostanie wpisane rozszerzenie, zostanie utworzona kopia zapasowa plików o rozszerzeniu rozszerzenie.

-l

Ustawia separator rekordów wyjściowych \$\ na taki sam jak separator rekordów wejściowych \$\ (zobacz -0), dzięki czemu do danych wysyłanych na wyjście przez metodę print automatycznie dodawane są znaki końca wiersza. Opcja ta jest przeznaczona do użytku z opcjami -p lub -n. W przypadku użycia jednej z tych opcji -l automatycznie wywołuje metodę chop usuwającą separator rekordów wejściowych z każdego wiersza danych wejściowych.

-n

Uruchamia program tak, jakby znajdował się w poniższej pętli:

```
while gets          # Wczytanie wiersza danych do zmiennej $_.  
  $F = split if $-a  # Rozbicie $_ na pola, jeśli została użyta opcja -a.  
  chop! if $-l        # Odcięcie końca wiersza z $_, jeśli użyto opcji -l.  
  # Tekst programu.  
end
```

Opcja ta działa w Ruby 1.9, mimo że funkcje globalne chop! i split nie są już dostępne w tej wersji języka.

Opcja ta jest często używana w połączeniu z opcją -e. Zobacz też opcję -p.

-p

Uruchamia program tak, jakby znajdował się w poniższej pętli:

```
while gets          # Wczytuje wiersz danych do $_.  
  $F = split if $-a  # Rozbija $_ na pola, jeśli została użyta opcja -a.  
  chop! if $-l        # Odcina zakończenie wiersza z $_, jeżeli została użyta opcja -l.  
  # Tekst programu.  
  print              # Wysyła na wyjście $_ (dodając $, jeśli użyto opcji -l).  
end
```

Opcja ta działa w Ruby 1.9, mimo że funkcje globalne `chop!` i `split` nie są już dostępne w tej wersji języka.

Opcja ta jest często używana z opcją `-e`. Zobacz także `-n`.

10.1.5. Różne opcje

Poniższe opcje nie pasują do żadnej z powyższych kategorii:

- c Analizuje tekst programu i zgłasza znalezione błędy składni bez uruchamiania go.
- C *katalog*, -X *katalog* Zmieniają aktualny katalog na katalog przed uruchomieniem programu.
- S Powoduje, że interpreter przetwarza wszystkie argumenty znajdujące się za nazwą programu i zaczynające się łącznikiem. Dla argumentów `-x=y` ustawia `$x` na `y`. Dla argumentów typu `-x` ustawia `$x` na wartość `true`. Wstępnie przetworzone argumenty są usuwane z tablicy ARGV.
- Tn Szuka określonego pliku programu względem ścieki określonej w zmiennej środowiskowej `RUBY_PATH`. Jeśli go tam nie znajdzie, szuka względem zmiennej środowiskowej `PATH`. W przypadku gdy nadal nic nie znajdzie, szuka w normalny sposób.
- Tx[*katalog*] Ustawia zmienną `$SAFE` na `n` lub `1`, jeśli `n` jest opuszczone. Więcej na ten temat znajduje się w podrozdziale 10.5.
- x[*katalog*] Wydobywa kod źródłowy Ruby z pliku programu, odrzucając wszystkie wiersze znajdujące się przed pierwszym wierszem zaczynającym się od znaków `#!ruby`. Dla zachowania zgodności z opcją `-X` (z wielkiej litery) niniejsza opcja pozwala także na określenie katalogu.

10.2. Środowisko najwyższego poziomu

W chwili uruchamiania interpreter Ruby definiuje i przygotowuje do użytku w programie pewną liczbę klas, modułów, stałych, zmiennych globalnych i funkcji globalnych. Kolejne podrozdziały zawierają ich zestawienie.

10.2.1. Predefiniowane moduły i klasy

Przy uruchamianiu interpretera Ruby 1.8 definiowane są następujące moduły:

Comparable	FileTest	Marshal	Precision
Enumerable	GC	Math	Process
Errno	Kernel	ObjectSpace	Signal

Klasy:

Array	File	Method	String
Bignum	Fixnum	Module	Struct
Binding	Float	NilClass	Symbol

Class	Hash	Numeric	Thread
Continuation	IO	Object	ThreadGroup
Data	Integer	Proc	Time
Dir	MatchData	Range	TrueClass
FalseClass	MatchingData	Regexp	UnboundMethod

Klasy wyjątków:

ArgumentError	NameError	SignalException
EOFError	NoMemoryError	StandardError
Exception	NoMethodError	SyntaxError
FloatDomainError	NotImplementedError	SystemCallError
IOError	RangeError	SystemExit
IndexError	RegexpError	SystemStackError
Interrupt	RuntimeError	ThreadError
LoadError	ScriptError	TypeError
LocalJumpError	SecurityError	ZeroDivisionError

Ruby 1.9 dodaje następujące moduły, klasy i wyjątki:

BasicObject	FiberError	Mutex	VM
Fiber	KeyError	StopIteration	

Aby sprawdzić predefiniowane moduły, klasy i wyjątki w swojej implementacji, można posłużyć się poniższym programem:

```
# Drukuj wszystkie moduły (z wyjątkiem klas).
puts Module.constants.sort.select { |x| eval(x.to_s).instance_of? Module}
# Drukuj wszystkie klasy (oprócz wyjątków).
puts Module.constants.sort.select { |x|
  c = eval(x.to_s)
  c.is_a? Class and not c.ancestors.include? Exception
}
# Drukuj wszystkie wyjątki.
puts Module.constants.sort.select { |x|
  c = eval(x.to_s)
  c.instance_of? Class and c.ancestors.include? Exception
}
```

10.2.2. Stałe najwyższego poziomu

Podczas uruchamiania interpretera definiowane są następujące stałe najwyższego poziomu (poza wymienionymi wcześniej modułami i klasami). W module definiującym stałą o takiej samej nazwie jak jedna z tych dostęp do stałych najwyższego poziomu można uzyskać za pomocą przedrostka `::`. Poniższy kod wyświetla listę stałych najwyższego poziomu:

```
ruby -e 'puts Module.constants.sort.reject{|x| eval(x.to_s).is_a? Module}'
```

ARGV

Obiekt klasy `IO` dający dostęp do wirtualnej konkatenacji plików wyznaczonych w tablicy `ARGV` lub do standardowego wejścia, jeśli tablica `ARGV` jest pusta. Synonim `$<`.

DATA

Jeżeli program zawiera wiersz, w którym znajduje się wyłącznie token `_END_`, stała ta jest strumieniem dającym dostęp do danych znajdujących się za tokenem `_END_`. W przypadku gdy program nie zawiera tokenu `_END_`, stała ta nie jest definiowana.

ENV

Obiekt zachowujący się jak tablica asocjacyjna i dający dostęp do ustawień zmiennych środowiskowych interpretera.

FALSE

Wycofywany synonim `false`.

NIL

Wycofywany synonim `nil`.

RUBY_PATCHLEVEL

Łańcuch określający numer ostatniej poprawki (ang. *patchlevel*) interpretera.

RUBY_PLATFORM

Łańcuch określający platformę interpretera.

Ruby_RELEASE_DATE

Łańcuch określający datę wydania interpretera.

RUBY_VERSION

Łańcuch określający wersję języka Ruby obsługiwana przez interpreter.

STDERR

Standardowy strumień wyjściowy błędów. Jest to domyślna wartość zmiennej `$stderr`.

STDIN

Standardowy strumień wejściowy. Jest to domyślna wartość zmiennej `$stdin`.

STDOUT

Standardowy strumień wyjściowy. Jest to domyślna wartość zmiennej `$stdout`.

TOPLEVEL_BINDING

Obiekt klasy `Binding` reprezentujący dowiązania w zakresie najwyższego poziomu.

TRUE

Wycofywany synonim `true`.

10.2.3. Zmienne globalne

Interpreter Ruby definiuje wstępnie kilka zmiennych globalnych, których można używać w programach. Wiele z nich ma pewne wyróżniające je cechy. Niektóre z nich mają w nazwach znaki interpunkcyjne (moduł `English.rb` definiuje angielskie odpowiedniki dla tych znaków interpunkcyjnych; aby móc używać tych bardziej rozwlekłych alternatyw, należy dodać do programu wiersz `require 'English'`). Część zmiennych jest przeznaczona tylko do odczytu. Inne są lokalne dla wątków, dzięki czemu każdy wątek może widzieć dla nich inną wartość. W końcu niektóre zmienne globalne (`$_`, `$~` i pochodne zmienne dopasowujące wzorce) są lokalne dla metod — mimo że zmieniona jest globalna, jej wartość jest lokalna dla bieżącej metody. Jeśli jakaś metoda ustawia wartość którejś z tych magicznych zmiennych globalnych, nie zmienia ona wartości widzianej przez kod, który wywołał tę metodę.

Pełną listę zmiennych globalnych predefiniowanych przez interpreter Ruby można uzyskać za pomocą poniższego kodu:

```
ruby -e 'puts global_variables.sort'
```

Aby do listy dodać dłuższe nazwy z modułu `English`, należy napisać:

```
ruby -rEnglish -e 'puts global_variables.sort'
```

Kolejne podrozdziały opisują te predefiniowane zmienne globalne według kategorii.

10.2.3.1. Ustawienia globalne

Te zmienne globalne przechowują ustawienia konfiguracyjne i określają informacje, na przykład argumenty wiersza poleceń na temat środowiska, w którym działa program:

- \$* Umożliwiający tylko odczyt synonim stałej ARGV. Angielski synonim — \$ARGV.
- \$\$ Identyfikator bieżącego procesu języka Ruby. Tylko do odczytu. Angielskie synonimy — \$PID i \$PROCESS_ID.
- \$? Stan wyjściowy ostatniego zakończonego procesu. Tylko do odczytu i lokalny dla wątku. Angielski synonim — \$CHILD_STATUS.
- \$DEBUG, \$-d Należy ustawić na wartość true, jeśli w wierszu poleceń użyto opcji -d lub --debug.
- \$KCODE, \$-K W Ruby 1.8 zmienna ta przechowuje łańcuch określający bieżące kodowanie tekstu. Jego wartością może być NONE, UTF-8, SJIS lub EUC. Wartość tę można ustawić za pomocą opcji interpretera -K. Zmienna ta nie działa w Ruby 1.9 i powoduje zgłaszanie ostrzeżenia.
- \$LOADED_FEATURES, \$" Tablica łańcuchów określających załadowane pliki. Tylko do odczytu.
- \$LOAD_PATH, \$:, \$-I Tablica łańcuchów przechowująca katalogi, które mają być przeszukiwane podczas ładowania plików za pomocą metod load i require. Zmienna ta jest tylko do odczytu, ale można zmodyfikować zawartość tablicy, do której się ona odwołuje, dodając na przykład na początku lub końcu nowe katalogi.
- \$PROGRAM_NAME, \$0 Nazwa pliku zawierającego kod źródłowy aktualnie wykonywanego programu. W przypadku gdy program jest wczytywany ze standardowego strumienia wejściowego, wartością będzie - lub e, jeśli program został wyznaczony z opcją -e. Zauważ, że to nie jest to samo co \$FILENAME.
- \$SAFE Bieżący bezpieczny poziom dla wykonywania programu. Szczegóły na ten temat zawiera podrozdział 10.5. Zmienną tę można ustawić w wierszu poleceń za pomocą opcji -T. Wartość tej zmiennej jest lokalna dla wątku.
- \$VERBOSE, \$-v, \$-w Jeśli opcje wiersza poleceń -v, -w lub --verbose zostały użyte, zmienne te mają wartość true. Jeżeli została użyta opcja -W0, wartością jest nil, w przeciwnym przypadku false. Można tę zmienną ustawić na nil, aby stłumić wszystkie ostrzeżenia.

10.2.3.2. Zmienne globalne obsługi wyjątków

Poniższe dwie zmienne globalne są przydatne w klauzulach `rescue`, kiedy zostaje zgłoszony wyjątek:

`$!`

Ostatni zgłoszony obiekt wyjątku. Do obiektu tego można też uzyskać dostęp w deklaracji klauzuli `rescue` za pomocą składni =>. Wartość tej zmiennej jest lokalna dla wątku. Angielski synonim — `$ERROR_INFO`.

`$@`

Dane stosu ostatniego wyjątku równoważne z `$!.backtrace`. Wartość ta jest lokalna dla wątku. Angielski synonim — `$ERROR_POSITION`.

10.2.3.3. Strumienie i zmienne globalne przetwarzania tekstu

Poniższe zmienne globalne są strumieniami i zmiennymi wejścia i wyjścia. Mają one wpływ na domyślne działanie metod modułu `Kernel` przetwarzających tekst. Przykłady ich użycia znajdują się w podrozdziale 10.3.

`$-`

Ostatni łańcuch wczytany przez metody `gets` i `readline` z modułu `Kernel`. Wartość ta jest lokalna dla wątku i metody. Kilka metod modułu `Kernel` niejawnie działa na zmiennej `$_`. Angielski synonim — `$LAST_READ_LINE`.

`$<`

Tylko do odczytu synonim strumienia `ARGF` — obiekt typu `IO` dający dostęp do wirtualnej konkatenacji plików wyznaczonych w wierszu polecień lub standardowego wejścia, jeśli żadne pliki nie zostały wyznaczone. Metody odczytujące modułu `Kernel`, na przykład `gets`, pobierają dane z tego strumienia. Należy zauważać, że strumień ten to nie zawsze to samo co `$stdin`. Angielski synonim — `$DEFAULT_INPUT`.

`$stdin`

Standardowy strumień wejściowy. Wartością początkową tej zmiennej jest stała `STDIN`. Wiele programów Ruby czyta ze strumienia `ARGF` lub `$<` zamiast ze zmiennej `$stdin`.

`$stdout, $>`

Standardowy strumień wyjściowy oraz cel metod drukujących modułu `Kernel` — `puts`, `print`, `printf` itd. Angielski synonim — `$DEFAULT_OUTPUT`.

`$stderr`

Standardowy strumień wyjściowy błędów. Wartość początkowa tej zmiennej jest stałą `STDERR`.

`$FILENAME`

Nazwa aktualnie odczytywanego pliku ze strumienia `ARGF`. Równoważna z `ARGF.filename`. Tylko do odczytu.

`$.`

Numer ostatniego wiersza wczytanego z aktualnego pliku wejściowego. Równoważna z `ARGF.lineno`. Angielskie synonimy — `$NR`, `$INPUT_LINE_NUMBER`.

`$/, $-`

Separator rekordów danych wejściowych (domyślnie znak nowego wiersza). Metody `gets` i `readline` używają tej wartości domyślnie do określania granicy wiersza. Wartość

tę można ustawić za pomocą opcji interpretera -O. Angielskie synonimy — \$RS, \$INPUT_RECORD_SEPARATOR.

\$\
Separator rekordów danych wyjściowych. Wartość domyślna to nil, ale jest ustawiana na \$/, jeśli zostanie użyta opcja wiersza poleceń -l. Jeżeli separator jest inny niż nil, jest on wysyłany po każdym wywołaniu metody print (ale nie po puts i innych metodach drukujących). Angielskie synonimy — \$ORS, \$OUTPUT_RECORD_SEPARATOR.

\$,
Separator wstawiany pomiędzy argumentami metody print, a także domyślny separator metody Array.join. Domyślnie ma on wartość nil. Angielskie synonimy — \$OFS, \$OUTPUT_FIELD_SEPARATOR.

\$;, \$-F
Domyślny separator pól używany przez metodę split. Domyślnie jest nil, ale wartość tę można zmienić za pomocą opcji interpretera -F. Angielskie synonimy — \$FS, \$FIELD_SEPARATOR.

\$F
Zmienna ta jest definiowana, jeśli interpreter zostanie wywołany z opcją -a i opcją -n lub -p. Przechowuje pola aktualnego wiersza wejściowego zwrócone przez metodę split.

10.2.3.4. Zmienne globalne dopasowywania wzorców

Poniższe zmienne globalne są lokalne dla wątku i metody i ustawia się je za pomocą operacji dopasowywania wzorców wyrażeń regularnych:

\$~
Obiekt klasy MatchData utworzony przez ostatnią operację dopasowywania wzorca. Wartość ta jest lokalna dla wątku i dla metody. Pozostałe zmienne dopasowywania wzorców opisane w niniejszym podrozdziale są pochodnymi tej zmiennej. Ustawienie tej zmiennej na nowy obiekt klasy MatchData modyfikuje wartość pozostałych wymienionych zmiennych. Angielski synonim — \$MATCH_INFO.

\$&
Ostatnio dopasowany tekst. Równoważna z \$~[0]. Tylko do odczytu, lokalna dla wątku, lokalna dla metody, pochodna zmiennej \$. Angielski synonim — \$MATCH.

\$`
Łańcuch znajdujący się przed dopasowaniem w ostatnim dopasowaniu wzorca. Równoważna z \$~.pre_match. Tylko do odczytu, lokalna dla wątku, lokalna dla metody, pochodna zmiennej \$. Angielski synonim — \$PREMATCH.

\$'
Łańcuch znajdujący się za dopasowaniem w ostatnim dopasowaniu wzorca. Równoważna z \$~.post_match. Tylko do odczytu, lokalna dla wątku, lokalna dla metody, pochodna zmiennej \$. Angielski synonim — \$POSTMATCH.

\$+
Łańcuch odpowiadający ostatnio dopasowanej grupie w ostatnim dopasowaniu wzorca. Tylko do odczytu, lokalna dla wątku, lokalna dla metody, pochodna zmiennej \$. Angielski synonim — \$LAST_PAREN_MATCH.

10.2.3.5. Zmienne globalne opcji wiersza poleceń

Ruby definiuje kilka zmiennych globalnych odpowiadających stanowi lub wartościom opcji wiersza poleceń. Zmienne \$-0, \$-F, \$-I, \$-K, \$-d, \$-v i \$-w mają synonimy i zostały opisane w poprzednich podrozdziałach.

\$-a

Wartość true, jeśli została użyta opcja interpretera -a. W przeciwnym przypadku false. Tylko do odczytu.

\$-i

Wartość nil, w sytuacji gdy nie została użyta opcja interpretera -i. W przeciwnym przypadku zmienna ta jest ustawiana na zapasowe rozszerzenie pliku określone za pomocą opcji -i.

\$-l

Wartość true, jeżeli została użyta opcja -l. Tylko do odczytu.

\$-p

Wartość true, gdy została użyta opcja interpretera -p. W przeciwnym przypadku false. Tylko do odczytu.

\$-W

W Ruby 1.9 zmienna ta określa aktualny poziom szczegółowości informacji. Ma wartość 0, jeśli została użyta opcja -W0, i 2, jeżeli została użyta opcja -w, -v lub --verbose. W przeciwnym przypadku ma wartość 1. Tylko do odczytu.

10.2.4. Predefiniowane funkcje globalne

Moduł Kernel, który jest dołączany do klasy Object, udostępnia kilka prywatnych metod egzemplarza służących jako funkcje globalne. Ponieważ są one prywatne, muszą być wywoływane tak jak funkcje, bez bezpośredniego obiektu odbiorcy. Natomiast dzięki temu że są dołączane do klasy Object, można je wywoływać wszędzie — self jest obiektem bez względu na wartość, a więc metody te mogą być wywoływane niejawnie na jego rzecz. Funkcje zdefiniowane w module Kernel można podzielić na kilka kategorii, z których większość została już opisana w tym rozdziale lub gdzieś indziej w tej książce.

10.2.4.1. Funkcje jak słowa kluczowe

Poniższe funkcje modułu Kernel zachowują się jak słowa kluczowe języka i zostały opisane w innych miejscach tej książki:

block_given?	iterator?	loop	require
callcc	lambda	proc	throw
catch	load	raise	

10.2.4.2. Wprowadzanie tekstu, wyprowadzanie tekstu i manipulowanie tekstem

Moduł Kernel definiuje następujące funkcje, z których większość to globalne wersje metod klasy IO. Ich szczegółowe opisy znajdują się w podrozdziale 10.3:

format	print	puts	sprintf
gets	printf	readline	
p	putc	readlines	

W Ruby 1.8 (w Ruby 1.9 nie) moduł Kernel definiuje też następujące wersje globalne metod klasy String, które działają niejawnie na zmiennej globalnej `$_`:

```
chomp    chop     gsub     scan    sub
chomp!   chop!   gsub!   split   sub!
```

10.2.4.3. Metody systemu operacyjnego

Poniższe funkcje modułu Kernel umożliwiają interakcję programu Ruby z systemem operacyjnym. Są one zależne od platformy, a ich opisy można znaleźć w podrozdziale 10.4. Należy zauważyć, że ` jest specjalną metodą zwracającą tekst wysłany przez dowolne polecenie powłoki systemu operacyjnego:

```
`      fork     select  system  trap
exec  open    syscall test
```

10.2.4.4. Ostrzeżenia, awarie i zamknięcie

Poniższe funkcje modułu Kernel wyświetlają ostrzeżenia, zgłaszą wyjątki, powodują zamknięcie programu lub rejestrują bloki kodu do wykonania przed zamknięciem programu. Zostały opisane razem z metodami systemów operacyjnych w podrozdziale 10.4:

```
abort   at_exit exit    exit!   fail    warn
```

10.2.4.5. Funkcje refleksji

Poniższe funkcje modułu Kernel wchodzą w skład API refleksyjnego Ruby i zostały opisane w rozdziale 8.:

binding	set_trace_func
caller	singleton_method_added
eval	singleton_method_removed
global_variables	singleton_method_undefined
local_variables	trace_var
method_missing	untrace_var
remove_instance_variable	

10.2.4.6. Funkcje konwersji

Poniższe funkcje modułu Kernel próbują konwertować swoje argumenty na nowe typy. Zostały opisane w podrozdziale 3.8.7.3:

```
Array  Float   Integer String
```

10.2.4.7. Różne funkcje modułu Kernel

Poniższe funkcje modułu Kernel nie pasują do żadnej z wymienionych kategorii:

autoload	rand	srand
autoload?	sleep	

Metody `rand` i `srand` generują liczby losowe i zostały opisane w podrozdziale 9.3.7. Metody `autoload` i `autoload?` zostały opisane w podrozdziale 7.6.3. Natomiast metoda `sleep` została opisana w podrozdziałach 9.9 i 10.4.4.

10.2.5. Funkcje globalne zdefiniowane przez użytkownika

Jeśli metoda zostanie zdefiniowana za pomocą słowa kluczowego `def` wewnątrz deklaracji `class` lub `module` i nie zostanie podany obiekt odbiorcy dla tej metody, zostanie ona publiczną metodą egzemplarza `self`, gdzie `self` jest definiowaną klasą lub definiowanym modelem. Użycie słowa kluczowego `def` na najwyższym poziomie, poza wszelkimi definicjami `class` i `module`, jest inne z dwóch ważnych powodów. Po pierwsze, metody najwyższego poziomu są metodami egzemplarza klasy `Object` (mimo że `self` nie jest obiektem klasy `Object`). Po drugie, metody najwyższego poziomu są zawsze prywatne.

Obiekt `self` najwyższego poziomu — obiekt `main`

Ponieważ metody najwyższego poziomu są metodami klasy `Object`, można się spodziewać, że wartością obiektu `self` jest `Object`. W rzeczywistości jednak metody najwyższego poziomu są specjalnym przypadkiem — są one definiowane w klasie `Object`, ale `self` jest innym obiektem. Ten specjalny obiekt najwyższego poziomu jest nazywany obiektem „`main`” i niewiele więcej można na jego temat powiedzieć. Klasą obiektu `main` jest klasa `Object`. Udostępnia on singletonową metodę `to_s`, która zwraca łańcuch „`main`”.

To, że metody najwyższego poziomu są definiowane w klasie `Object`, oznacza, że są dziedziczone przez wszystkie obiekty (w tym przez klasy `Module` i `Class`) i (jeśli nie zostaną przesłonięte) można ich używać we wszystkich definicjach metod klasowych i egzemplarza. Aby się co do tego upewnić, można przypomnieć sobie algorytm rozstrzygania nazw metod opisany w podrozdziale 7.8. To, że metody najwyższego poziomu są prywatne, oznacza, że muszą być wywoływanie jak funkcje — bez bezpośrednio określonego odbiorcy. W ten sposób Ruby naśladuje paradygmat programowania proceduralnego w swoich ściśle obiektowych ramach.

10.3. Praktyczne skróty do ekstrakcji i raportowania

W języku Ruby widoczne są wpływy języka skryptowego Perl, którego nazwa jest akronimem angielskich słów *Practical Extraction and Reporting Language* (praktyczny język ekstrakcji i raportowania). Z tego powodu Ruby udostępnia kilka funkcji ułatwiających pisanie programów wydobywających dane z plików i generujących raporty. W paradygmacie programowania obiektowego funkcje wejścia i wyjścia są metodami klasy `I/O`, a funkcje do manipulowania łańcuchami tekstu są metodami klasy `String`. Jednak z pragmatycznych powodów dobrze jest mieć globalne funkcje odczytujące z predefiniowanych strumieni wejściowych i wyjściowych i do nich zapisujące. Poza udostępnieniem tych globalnych funkcji język Ruby definiuje dla nich także specjalne zachowania — wiele z nich operuje na specjalnej zmiennej lokalnej w zasięgu metod o nazwie `$_`. Zmienna ta przechowuje ostatni wiersz wczytany ze strumienia wejściowego. Znak podkreślenia jest mnemonikiem — wygląda jak linia (większość zmiennych globalnych języka Ruby, w których nazwach znajdują się znaki interpunkcyjne, została odziedziczona po języku Perl). Poza globalnymi funkcjami wejścia i wyjścia dostępnych jest też kilka globalnych funkcji przetwarzania łańcuchów tekstu, które działają jak metody klasy `String`, ale działają niejawnie na zmiennej `$_`.

Te globalne funkcje i zmienne pełnią rolę skrótów używanych w krótkich i prostych skryptach Ruby. Używanie ich w większych programach jest ogólnie uznawane za zły styl.

10.3.1. Funkcje wejściowe

Funkcje globalne `gets`, `readline` i `readlines` są takie same jak metody o takich samych nazwach zdefiniowane w klasie `IO` (zobacz podrozdział 9.7.3.1), ale działają niejawnie na strumieniu `$<` (który jest dostępny także jako stała o nazwie `ARGV`). Podobnie jak metody klasy `IO`, te funkcje globalne niejawnie ustawiają wartość zmiennej `$_`.

Strumień `$<` zachowuje się jak obiekt klasy `IO`, ale nie jest takim obiektem (jego metoda `class` zwraca obiekt klasy `Object`, a metoda `to_s` łańcuch "ARGV"). Zachowanie tego strumienia jest skomplikowane. Jeśli tablica `ARGV` jest pusta, strumień `$<` jest taki sam jak `STDIN` — standardowy strumień wejściowy. Jeżeli strumień `ARGV` nie jest pusty, Ruby zakłada, że jest to lista nazw plików. W takim przypadku strumień `$<` zachowuje się, jakby odczytywał z konkatenacji każdego z tych plików. Nie jest to jednak dokładne zachowanie strumienia `$<`. Przy pierwszym żądaniu odczytu ze strumienia `$<` Ruby usuwa z tablicy `ARGV` pierwszą nazwę pliku za pomocą metody `ARGV.shift`. Otwiera ten plik i odczytuje jego dane. Kiedy dojdzie do końca pliku, powtarza cały proces, usuwając kolejną nazwę pliku z tablicy `ARGV` i otwierając ten plik. Strumień `$<` nie zgłasza końca pliku, dopóki w tablicy `ARGV` są kolejne pliki.

Oznacza to, że skrypty Ruby mogą modyfikować zawartość tablicy `ARGV` (na przykład aby przetworzyć opcje wiersza poleceń) przed rozpoczęciem odczytu ze strumienia `$<`. Skrypt może też dodawać pliki do tablicy `ARGV` w trakcie działania i strumień `$<` używa ich.

10.3.2. Wycofywane funkcje ekstrakcji

W Ruby do wersji 1.8 funkcje globalne `chomp`, `chomp!`, `chop`, `chop!`, `gsub`, `gsub!`, `scan`, `split`, `sub` i `sub!` działają tak samo jak metody o tych samych nazwach w klasie `String`, ale operują niejawnie na zmiennej `$_`. Ponadto funkcje `chomp`, `chop`, `gsub` i `sub` przypisują swoje wyniki z powrotem do zmiennej `$_`, co oznacza, że są synonimami swoich wersji z wykryznikiem.

Funkcje te zostały usunięte z Ruby 1.9, a więc nie należy już ich używać w nowych programach.

10.3.3. Funkcje raportujące

Moduł `Kernel` definiuje kilka funkcji globalnych wysyłających dane do zmiennej globalnej `$stdout` (zmienna ta początkowo odnosi się do standardowego strumienia wyjściowego `STDOUT` procesu Ruby, ale można zmienić jej wartość i zachowanie opisanych tu funkcji).

Funkcje `puts`, `print`, `printf` i `putc` są odpowiednikami metod o tych samych nazwach strumienia `STDOUT` (zobacz podrozdział 9.7.4). Przypomnijmy, że metoda `puts` dodaje do swoich danych wyjściowych znak nowego wiersza, jeśli go tam nie ma. Metoda `print` natomiast nie dodaje znaku nowego wiersza, ale dodaje separator rekordów wyjściowych `$\`, jeśli ta zmienna globalna została ustawiona.

Funkcja globalna `p` jako jedyna nie ma odpowiednika w klasie `IO`. Jest przeznaczona do debugowania, a jej krótka nazwa bardzo ułatwia jej wpisywanie. Wywołuje metodę `inspect` każdego ze swoich argumentów i przekazuje powstałe łańcuchy do metody `puts`. Przypomnijmy, że domyślnie `inspect` jest równoważna z metodą `to_s`, ale niektóre klasy przeddefiniują ją, aby wyświetlać bardziej przyjazne programistycie dane podczas debugowania. Po dołączeniu do programu biblioteki `pp` zamiast funkcji `p` można używać funkcji `pp` „ładnie drukującej” dane debugera (przydaje się ona do drukowania dużych tablic jednowymiarowych i asocjacyjnych).

Wspomniana wcześniej metoda `printf` przyjmuje łańcuch formatu jako pierwszy argument i wstawia wartości swoich pozostałych argumentów do tego łańcucha przed wysłaniem na wyjście wyniku. łańcuch ten można także sformatować bez wysyłania rezultatu do zmiennej `$stdout` za pomocą funkcji globalnej `sprintf` lub jej synonimu `format`. Działają one jak operator `%` z klasy `String`.

10.3.4. Skróty w skryptach jednowierszowych

Wczesniej w niniejszym rozdziale opisana została opcja interpretera `-e` wykonująca pojedynczy wiersz kodu Ruby (często używana w połączeniu z opcjami pętlowymi `-n` i `-p`). Istnieje jeden specjalny skrót dziedziczący po języku Perl, którego można używać wyłącznie w skryptach wykonywanych za pomocą opcji `-e`.

Jeśli skrypt jest podany za pomocą opcji `-e` i literał wyrażenia regularnego znajduje się jako jedyny w wyrażeniu warunkowym (jest częścią instrukcji albo modyfikatora `if`, `unless`, `while` lub `until`), wyrażenie to jest niejawnie porównywane z zawartością zmiennej `$_`. Aby na przykład wydrukować wszystkie linijki danych znajdujące się w pliku, które zaczynają się od litery A, można napisać:

```
ruby -n -e 'print if /^A/' datafile
```

Gdyby ten sam skrypt był zapisany w pliku i został uruchomiony bez opcji `-e`, nadal by działał, ale zgłaszałby ostrzeżenie (nawet bez `-w`). Aby uniknąć tego ostrzeżenia, trzeba by było to porównywanie uczynić jawnym:

```
print if $_ =~ /(^A/
```

10.4. Wywoływanie systemu operacyjnego

Ruby udostępnia kilka funkcji globalnych komunikujących się z systemem operacyjnym w celu wykonywania programów, tworzenia nowych procesów, obsługi sygnałów itd. Początkowo Ruby był tworzony z myślą o systemach uniksowych i wiele z tych funkcji związanych z systemem operacyjnym odzwierciedlało to dziedzictwo. Z natury funkcje te nie są tak przenośne jak pozostałe funkcje, a niektórych z nich nie można w ogóle zaimplementować w systemie Windows i innych nieuniksowych platformach. Kolejne podrozdziały opisują niektóre najczęściej używane z tych zależnych od systemu funkcji. Funkcje typu `syscall`, które działają na szczególnie niskim poziomie i są zależne od platformy, nie zostały opisane.

10.4.1. Wywoływanie poleceń systemu operacyjnego

Metoda `Kernel.``` przyjmuje jeden argument reprezentujący polecenie powłoki systemu operacyjnego. Uruchamia podpowłokę i przekazuje do niej dane polecenie. Wartością zwrotną jest tekst drukowany w standardowym strumieniu wyjściowym. Metoda ta jest zazwyczaj wywoływana przy użyciu specjalnej składni — na literałach łańcuchowych otoczonych dwiema apostrofami lub literałach łańcuchowych ograniczonych znakami `%%` (zobacz podrozdział 3.2.1.6). Na przykład:

```
os = `uname`          # Literal łańcuchowy i wywołanie metody w jednym.  
os = %x{uname}       # Inna składnia.  
os = Kernel.``("uname") # Bezpośrednie wywołanie metody.
```

Metoda ta nie wywołuje po prostu określonego polecenia — wywołuje powłokę, co oznacza, że dostępne są takie własności powłoki jak rozwijanie znaków specjalnych w nazwach plików:

```
files = `echo *.xml`
```

Innym sposobem na uruchomienie procesu i odczytanie jego danych jest użycie funkcji `Kernel.open`. Metoda ta jest wersją metody `File.open` i najczęściej używa się jej do otwierania plików (jeśli dodatkowo umieści się w programie wiersz `require 'open-uri'`, można za jej pomocą otwierać także adresy HTTP i FTP). Jeśli jednak pierwszym znakiem wyznaczonej nazwy pliku jest pionowa kreska `|`, w zamian zostaje otwarty kanał do odczytu i zapisu w określonym poleceniu powłoki:

```
pipe = open("|echo *.xml")
files = pipe.readline
pipe.close
```

Aby wywołać polecenie w powłoce, nie pobierając wyniku tego wywołania, należy użyć metody `Kernel.system`. Kiedy zostanie do niej przekazany łańcuch, wykonuje go ona w powłoce, czeka na zakończenie wykonywania tego polecenia i zwraca wartość `true` w razie powodzenia albo `false` w razie niepowodzenia operacji. Jeśli do metody `system` zostanie przekazanych kilka argumentów, pierwszy z nich jest nazwą programu, który ma zostać wywołany, a pozostałe argumentami wiersza poleceń. W takim przypadku na tych argumentach nie jest wykonywane rozwijanie powłoki.

Metodą działającą na niższym poziomie, która umożliwia wywoływanie programów, jest użycie funkcji `exec`. Funkcja ta nigdy nie zwraca wartości — zastępuje aktualnie działający proces Ruby innym wyznaczonym procesem. Może być użyteczna przy pisaniu skryptu Ruby opakowującego inny program do uruchomienia. Zazwyczaj jednak jest ona używana w połączeniu z funkcją `fork`, która została opisana w kolejnym podrozdziale.

10.4.2. Rozgałęzianie procesów

W podrozdziale 9.9 opisane zostało API do pisania programów wielowątkowych. Innym sposobem na programowanie współbieżne w języku Ruby jest użycie kilku procesów Ruby. Służy do tego funkcja `fork` lub jej synonim `Process.fork`. Najprościej jest użyć tej funkcji z blokiem:

```
fork {
  puts "Witajcie z procesu potomnego: $$$"
}
puts "Witajcie z procesu nadziednego: $$$"
```

Oryginalny proces kontynuuje wykonywanie kodu znajdującego się za blokiem, a nowy proces Ruby wykonuje kod znajdujący się w bloku.

Funkcja `fork` wywołana bez bloku zachowuje się inaczej. Wywołanie jej w procesie nadziednim powoduje zwrócenie liczby całkowitej wyznaczającej identyfikator nowo utworzonego procesu-potomka. W procesie-potomku to samo wywołanie funkcji `fork` zwraca wartość `nil`. W związku z tym powyższy kod można też zapisać następująco:

```
pid = fork
if (pid)
  puts "Proces nadziedny: $$$"
  puts "Utworzono proces podziedny #{pid}"
else
  puts Witajcie z procesu podziednegego: $$$"
end
```

Jedną z najważniejszych różnic pomiędzy procesami a wątkami jest to, że procesy nie współdzielą pamięci. W chwili wywołania funkcji `fork` zostaje uruchomiony nowy proces Ruby będący wierną kopią procesu nadziedzkiego. Jednak wszelkie zmiany stanu wykonywane przez tę funkcję (poprzez tworzenie lub modyfikowanie obiektów) są dokonywane w jej własnej przestrzeni adresowej. Potomek nie może zmodyfikować struktury rodzica, ani rodzic struktury potomka.

Aby umożliwić procesom rodzica i potomka wzajemną komunikację, należy użyć metody `open` i przekazać do niej jako pierwszy argument łańcuch " | - ". W ten sposób otwiera się kanał do ostatnio odgałęzionego procesu. Metoda `open` przekazuje sterowanie do odpowiedniego bloku zarówno w rodzicu, jak i w potomku. W tym drugim bloku odbiera wartość `nil`. Natomiast w rodzicu do bloku przekazywanego jest obiekt klasy `IO`. Z tego obiektu można odczytać dane pochodzące od potomka. Natomiast dane zapisywane w tym obiekcie są dostępne do odczytu w standardowym wejściu potomka. Na przykład:

```
open("|-", "r+") do |child|
  if child
    # Proces-rodzic.
    child.puts("Witaj dziecko")          # Wysyłanie do potomka.
    response = child.gets               # Odczyt z potomka.
    puts "Potomek mówi: #{response}"
  else
    # Proces-potomek.
    from_parent = gets                 # Odczyt z rodzica.
    STDERR.puts "Rodzic mówi: #{from_parent}"
    puts("Cześć mama!")                # Wysyłanie do rodzica.
  end
end
```

W połączeniu z funkcją `fork` lub metodą `open` przydatna jest funkcja `Kernel.exec`. Wcześniej dowiedziałeś się, że za pomocą funkcji ``` i `system` można wysłać dowolne polecenie do powłoki systemowej. Obie te metody są jednak synchroniczne — nie zwracają wartości, dopóki nie zakończy się wykonywanie polecenia. Aby wykonać polecenie systemu operacyjnego jako osobny proces, najpierw należy za pomocą funkcji `fork` utworzyć nowy proces, a następnie w tym procesie wywołać funkcję `exec` wykonującą wybrane polecenie. Wywołanie funkcji `exec` nigdy nie zwraca wartości — aktualny proces zostaje zastąpiony nowym. Argumenty funkcji `exec` są takie same jak funkcji `system`. Jeśli jest tylko jeden, jest on traktowany jako polecenie powłoki. Jeśli jest kilka argumentów, pierwszy z nich wyznacza, co ma zostać wykonane, a pozostałe stają się tablicą `ARGV` tego polecenia:

```
open("|-", "r") do |child|
  if child
    # Proces-rodzic.
    files = child.readlines      # Odczyt danych z potomka.
    child.close
  else
    # Proces-potomek.
    exec("/bin/ls", "-l")        # Uruchomienie czegoś jeszcze.
  end
end
```

Praca w wielu procesach jest programowaniem niskiego poziomu i szczegóły tej techniki wychodzą poza zakres niniejszej książki. Aby dowiedzieć się więcej na ten temat, najlepiej poczytać o innych metodach modułu `Process` przy użyciu narzędzia `ri`.

10.4.3. Przechwytywanie sygnałów

Większość systemów operacyjnych pozwala na asynchroniczne wysyłanie sygnałów do działających procesów. Ma to miejsce na przykład, kiedy użytkownik naciśnie kombinację klawiszy *Ctrl+C*, aby anulować działanie jakiegoś programu. Większość programów powłoki wysyła sygnał o nazwie **SIGINT** (przerywający) w odpowiedzi na naciśnięcie klawiszy *Ctrl+C*. Domyślną odpowiedzią na ten sygnał jest zazwyczaj przerwanie programu. Ruby pozwala na „przechwytywanie” tych sygnałów w programach i definiowanie własnych procedur ich obsługi. Służy do tego metoda `Kernel trap` (lub jej synonim `Signal trap`). Żeby na przykład uniemożliwić przerwanie działania programu za pomocą klawiszy *Ctrl+C*:

```
trap "SIGINT" {  
    puts "Ignoruję sygnał SIGINT."  
}
```

Zamiast bloku do metody `trap` można przekazać obiekt klasy `Proc`. Aby tylko po cichu zignorować sygnał, jako drugi argument można przekazać łańcuch `"IGNORE"`. Chcąc przywrócić domyślne zachowanie systemu operacyjnego, należy przekazać łańcuch `"DEFAULT"`.

W dugo działających programach jak serwery przydatne może być zdefiniowanie procedur obsługi sygnałów w taki sposób, by zmuszały serwer do ponownego sprawdzenia pliku konfiguracyjnego, zapisywały statystyki użycia w pliku dziennika czy przestawiały serwer w tryb debugowania. W systemach uniksowych do tego celu powszechnie wykorzystuje się sygnały `SIGUSR1` i `SIGUSR2`.

10.4.4. Zamykanie programu

W module `Kernel` znajduje się kilka powiązanych ze sobą metod służących do zamykania programów lub wykonywania podobnych działań. Najprostszą z nich jest metoda `exit`. Zgłasza ona wyjątek `SystemExit`, który, jeśli nie zostanie przechwycony, powoduje zamknięcie programu. Jednak przed zamknięciem wykonywane są wszystkie bloki `END` i procedury obsługi zamykania programu zarejestrowane za pomocą metody `Kernel.at_exit`. Aby zamknąć program natychmiast, należy użyć metody `exit!`. Obie wymienione metody przyjmują jako argument liczbę całkowitą określającą kod wyjściowy procesu raportowanego do systemu operacyjnego. Metody `Process.exit` i `Process.exit!` są synonimami tych funkcji modułu `Kernel`.

Funkcja `abort` drukuje wyznaczony komunikat o błędzie w standardowym strumieniu wyjściowym, a następnie wywołuje funkcję `exit(1)`.

Funkcja `fail` jest synonimem funkcji `raise`, a jej przeznaczeniem są przypadki, w których wyjątek ma spowodować zamknięcie programu. Funkcja `fail`, podobnie jak `abort`, wyświetla komunikat w chwili zamknięcia programu. Na przykład:

```
fail "Nieznana opcja #{switch}"
```

Funkcja `warn` jest spokrewniona z funkcjami `abort` i `fail` — drukuje ostrzeżenie w standardowym strumieniu błędów (jeśli ostrzeżenia nie zostały wyłączone za pomocą opcji `-W0`). Należy jednak zauważyć, że funkcja ta nie zgłasza wyjątku ani nie powoduje zamknięcia programu.

Inną podobną funkcją niezamykającą programu jest `sleep`. Powoduje ona wstrzymanie programu (lub przynajmniej bieżącego wątku) na określzoną liczbę sekund.

10.5. Bezpieczeństwo

W systemie języka Ruby dostępny jest mechanizm pozwalający pisać programy działające na niezaufanych danych i niezaufanym kodzie. System zabezpieczeń składa się z dwóch części. Pierwsza to mechanizm odróżniający bezpieczne dane od danych niebezpiecznych, czyli niepewnych (ang. *tainted*). Druga część to technika **ograniczonego środowiska wykonawczego**, która pozwala na zamknięcie środowiska Ruby i uniemożliwia interpreterowi wykonanie potencjalnie niebezpiecznych operacji na niepewnych danych. Pozwala to uniknąć ataków typu SQL injection, które polegają na podaniu złośliwego kodu zmieniającego działanie programu. Ograniczone środowisko wykonywania jest czymś więcej, ponieważ pozwala na wykonywanie niezaufanego (i możliwe, że złośliwego) kodu bez obawy, że usunie on pliki, ukradnie dane lub spowoduje jakieś inne szkody.

10.5.1. Dane niepewne

Każdy obiekt w języku Ruby jest pewny lub niepewny. Wartości literalne w kodzie programu są pewne. Natomiast wartości pochodzące ze środowisk zewnętrznych są niepewne. Zaliczają się do nich łańcuchy wczytywane z wiersza poleceń (ARGV), zmienne środowiskowe (ENV), a także wszelkie dane wczytywane z plików, gniazd lub innego rodzaju strumieni. Specjalnym przypadkiem jest zmienna środowiskowa PATH — jest niepewna tylko wówczas, gdy jeden lub więcej znajdujących się w niej katalogów pozwala na nieograniczone zapisywanie w nich danych. Co ważne, stan niepewności jest zaraźliwy, a więc obiekty pochodzące od obiektów niepewnych również są niepewne.

Metody klasy Object `taint`, `tainted?` i `untaint` pozwalają na oznaczenie pewnego obiektu jako niepewnego, sprawdzenie, czy dany obiekt jest niepewny, oraz oznaczenie niepewnego obiektu jako pewnego. Niepewny obiekt należy oznaczyć jako pewny wyłącznie wtedy, gdy zostanie on sprawdzony przez program i uznany za bezpieczny mimo jego niepewnego pochodzenia.

10.5.2. Ograniczone środowisko wykonawcze i poziomy zabezpieczeń

Ruby może wykonywać programy z włączonym **sprawdzaniem bezpieczeństwa**. Zmienna globalna `$SAFE` określa poziom zabezpieczeń. Domyślny poziom zabezpieczeń to 0, ale dla programów działających z uprawnieniami `setuid` lub `setgid` (są to uniksowe określenia programów działających z uprawnieniami wykraczającymi poza uprawnienia użytkownika, który je wywołał) wynosi 1. Dostępne poziomy zabezpieczeń to liczby całkowite 0, 1, 2, 3, 4. Poziom zabezpieczeń można ustawić za pomocą opcji interpretera `-T`. Można to także zrobić poprzez przypisanie do zmiennej `$SAFE`. Należy jednak zauważyć, że wartość tę można tylko zwiększyć — zmniejszenie jej jest niemożliwe:

```
$SAFE=1           # Zwiększenie poziomu zabezpieczeń.  
$SAFE=4           # Ponowne zwiększenie poziomu zabezpieczeń.  
$SAFE=0           # SecurityError! Tego nie wolno robić.
```

Zmienna \$SAFE jest lokalna dla wątku. Innymi słowy, można zmieniać jej wartość w jednym wątku, nie zmieniając jej w innych wątkach. Przy użyciu tej funkcji wątki można wykonywać w ograniczonym środowisku wykonawczym:

```
Thread.start {  
    $SAFE = 4  
    ...  
}  
      # Utworzenie wątku pełniącego funkcję ograniczonego środowiska wykonawczego.  
      # Ograniczenie wykonywania tylko w tym wątku.  
      # Niezaufany kod może być wykonywany tutaj.
```

Niniejszy opis poziomów zabezpieczeń odnosi się tylko do implementacji referencyjnej języka Ruby. Inne implementacje mogą się pod tym względem różnić. Na przykład bardzo mało w kierunku naśladowania ograniczonych trybów wykonywania implementacji referencyjnej (w chwili pisania tej książki) robi implementacja JRuby. Ponadto należy pamiętać, że model bezpieczeństwa języka Ruby nie został poddany tak szczegółowym i drobiazgowym badaniom jak architektura bezpieczeństwa Javy. Kolejne podrozdziały opisują, jak w założeniu ma działać ograniczone wykonywanie w języku Ruby, ale nieodkryte jeszcze błędy mogą umożliwić obejście tych zabezpieczeń.

10.5.2.1. Poziom zabezpieczeń 0

Poziom 0 jest domyślnym poziomem zabezpieczeń. Na tym poziomie niepewne dane nie są sprawdzane w żaden sposób.

10.5.2.2. Poziom zabezpieczeń 1

Na tym poziomie zabronione są potencjalnie niebezpieczne operacje przy użyciu niepewnych danych. Nie można wykonywać łańcucha kodu, jeśli jest on niepewny, nie można dołączyć do programu biblioteki, jeśli jest niepewna, nie można otworzyć niepewnego pliku ani połączyć się z niepewnym hostem. Programy, zwłaszcza serwery podłączone do sieci, przyjmujące dowolne dane powinny najprawdopodobniej używać tego poziomu. Pozwala on na wyłapywanie błędów programistycznych wykorzystujących w niebezpieczny sposób niepewne dane.

Pisząc bibliotekę wykonującą potencjalnie niebezpieczne operacje — na przykład łączenie się z serwerem baz danych — należy sprawdzić wartość zmiennej \$SAFE. Jeśli jej wartość wynosi 1 lub więcej, biblioteka ta nie powinna operować na obiektach niepewnych. Na przykład nie powinno się wysyłać zapytania SQL do bazy danych, jeżeli łańcuch zawierający to zapytanie jest niepewny.

Ograniczenia wykonawcze pierwszego poziomu zabezpieczeń są następujące:

- Zmienne środowiskowe RUBYLIB i RUBYOPT są ignorowane przy starcie.
- Bieżący katalog(.) nie jest dodawany do zmiennej \$LOAD_PATH.
- Zabronione jest używanie następujących opcji wiersza poleceń: -e, -i, -I, -r, -s, -S oraz -X.
- Zabronione jest używanie niektórych metod klasowych i egzemplarza klas Dir, IO, File i FileTest.
- Metod test, eval, require, load i trap nie można wywoływać z niepewnymi argumentami.

10.5.2.3. Poziom zabezpieczeń 2

Drugi poziom zabezpieczeń ogranicza działania na niepewnych danych, podobnie jak poziom pierwszy, ale nakłada także dodatkowe ograniczenia na sposoby manipulowania plikami i procesami bez względu na to, czy są pewne, czy nie. Nie ma zbyt wielu powodów, aby program sam ustawił taki poziom zabezpieczeń. Natomiast może to zrobić administrator systemu, który chce mieć pewność, że program nie utworzy ani nie usunie żadnych katalogów, nie zmieni pozwoleń dostępu do plików, nie wykona żadnych plików wykonywalnych, nie załaduje kodu Ruby z ogólnodostępnych katalogów itd.

Metody ograniczone na tym poziomie są następujące:

Dir.chdir	File.truncate	Process.egid=
Dir.chroot	File.umask	Process.fork
Dir.mkdir	IO.fctrl	Process.kill
Dir.rmdir	IO.ioctl	Process.setpgid
File.chmod	Kernel.exit!	Process.setpriority
File.chown	Kernel.fork	Process.setsid
File.flock	Kernel.syscall	
File.lstat	Kernel trap	

Dodatkowo drugi poziom zabezpieczeń uniemożliwia ładowanie lub dołączanie kodu Ruby lub wykonywanie czegokolwiek, co znajduje się w ogólnodostępnych katalogach.

10.5.2.4. Poziom zabezpieczeń 3

Trzeci poziom zabezpieczeń posiada wszystkie ograniczenia poziomu drugiego, a dodatkowo wszystkie obiekty — w tym literaly w kodzie źródłowym programu (ale wyłączając predefiniowane obiekty w środowisku globalnym) — są oznaczane jako niepewne w trakcie ich tworzenia. Ponadto zabronione jest używanie metody `untaint`.

Trzeci poziom jest pośrednim krokiem do poziomu czwartego i nie jest często używany.

10.5.2.5. Poziom zabezpieczeń 4

Ten poziom rozszerza poziom trzeciego, uniemożliwiając wszelkie modyfikacje pewnych obiektów (wliczając wywoływanie metody `taint` na ich rzecz). Program działający na tym poziomie nie może zmodyfikować globalnej zmiennej środowiskowej ani żadnych pewnych obiektów utworzonych wcześniej na niższym poziomie zabezpieczeń. W ten sposób tworzono jest ograniczone środowisko wykonawcze (ang. *sandbox*), w którym niezaufany kod może działać bez możliwości spowodowania szkód (przynajmniej teoretycznie — w przyszłości mogą zostać odkryte błędy w implementacji lub niedostatki w leżących u podłożu tego modelu zabezpieczeńach).

Wywoływanie metody `eval` na rzecz niepewnych łańcuchów jest zabronione na poziomach 1, 2 i 3. Na poziomie 4 jest znowu dozwolone, ponieważ inne ograniczenia na tym poziomie są na tyle silne, że taki łańcuch nie może spowodować żadnych szkód. Poniżej przedstawiony jest sposób wykonania dowolnego kodu na poziomie 4:

```
def safe_eval(str)
  Thread.start {
    $SAFE = 4
    eval(str)
  }.value
end
```

Uruchomienie wątku sandboks.
Zwiększenie poziomu zabezpieczeń.
Funkcja eval w sandboksie.
Pobranie wyniku.

Na czwartym poziomie zabezpieczeń nie można używać metody `require` do ładowania plików z kodem Ruby. Można używać metody `load`, ale tylko w opakowanej postaci z wartością `true` jako drugim argumentem. Dzięki temu Ruby umieści taki plik w sandboksie będącym anonimowym modułem, co uniemożliwi jego klasom, modułom i stałym zmodyfikowanie globalnej przestrzeni nazw. Oznacza to, że kod działający na czwartym poziomie zabezpieczeń może ładować, ale nie może używać klas i modułów zdefiniowanych w zewnętrznych modułach.

Poziom czwarty można dodatkowo ograniczyć, umieszczając wątek sandboksu (przed ustawieniem zmiennej `$SAFE`) w obiekcie klasy `ThreadGroup` i wywołując na rzecz tej grupy metodę `close`. Szczegóły na ten temat można znaleźć w podrozdziale 9.9.5.

Poziom czwarty uniemożliwia także wykonywanie następujących działań:

- wywoływanie metod `require`, `autoload` i `include` oraz nieopakowane wywoływanie metody `load`,
- modyfikowanie klasy `Object`,
- modyfikowanie pewnych klas lub modułów,
- wywoływanie metod metaprogramowania,
- manipulowanie innymi wątkami niż bieżący,
- dostęp do danych lokalnych wątku,
- zamknięcie procesów,
- pobieranie i wysyłanie plików,
- modyfikowanie zmiennych środowiskowych,
- inicjowanie generatora liczb losowych za pomocą metody `srand`.

-, 101
!, 16, 35, 105
!~, 104
!=, 58, 104
#, 32
coding:, 42
\$, 16, 35
\$', 365
\$!, 364
\$!.backtrace, 364
\$", 363, 365
\$\$, 363
\$\&, 365
\$*, 327, 363
\$\,, 365
\$\., 364
\$\/, 330, 364
\$\:, 237, 363
\$\;, 365
\$\?, 363
\$\@, 364
\$_, 329, 364
\$\~, 365
\$\+, 365
\$\<, 327, 364
\$\>, 364
\$\\$, 363
\$\-0, 364
\$\-a, 366
\$\-d, 363
\$DEBUG, 363
\$ERROR_POSITION, 364
\$F, 365
\$-F, 365
\$FILENAME, 364
\$-i, 366
\$-I, 363
\$-K, 363
\$KCODE, 363
\$-l, 366
\$LOAD_PATH, 237, 239, 363
\$LOADED_FEATURES, 363
\$MATCH_INFO, 365
\$-p, 366
\$PREMATCH, 365
\$PROGRAM_NAME, 363
\$SAFE, 84, 237, 344, 356, 363, 374
\$stderr, 327, 364
\$stdin, 327, 364
\$stdout, 327, 364
\$-v, 363
\$VERBOSE, 363
\$-w, 363
\$-W, 366
%, 48
%q, 54
%Q, 54
&, 68, 102, 113, 180
&&, 105
(, 112
, 15, 58, 113
**, 49, 101
. , 112
.. , 107
..., 107
; , 113
:: , 88, 112
; , 112
? , 35

?:, 109, 124
@, 16, 17, 35
@@, 16, 36
[], 15, 91, 112, 197, 198, 209
[]=, 15, 198, 313
^, 102
__ENCODING__, 43, 87
__END__, 36
__FILE__, 86, 259
__LINE__, 87, 259
__method__, 260
__send__, 221, 255
__END__, 41, 44
id, 75
_id2ref, 261
_index, 304
{}, 112
|, 68, 102
||, 105, 106
||=, 95
~, 102
+, 15, 57, 101
<, 103, 113
<<, 15, 19, 55, 58, 76, 102, 318
<<-, 55
<=, 103
<=>, 70, 71, 79, 103, 212
=, 16, 35
=~ , 79, 104, 290
==, 58, 77, 104, 209, 210, 221, 309
====, 17, 78, 104, 122, 235, 249
=>, 112
=begin, 32
=begin rdoc, 33
=end, 32
>, 103
>=, 103
>>, 102

A

aborting, 346
adresat, 166
after, 262
akcesory, 205
alias, 171
aliasy metod, 171
alive?, 164

allocate, 227
and, 100, 105
AND, 102, 103
anulowanie propagacji wyjątku, 156
apply, 194
ARGF, 327, 329, 361
ArgumentError, 153
argumenty blokowe, 178
argumenty metod, 174
argumenty wątkien, 161
ARGV, 327, 361, 374
ARGV.shift, 369
arity, 99, 185
Array, 13, 81, 129, 306
Array.new, 66, 307
Array.pack, 230, 284
Array.sort, 137
arytmetyka, 48
liczby dziesiętne, 296
ASCII, 41, 63
ascii_only?, 62
ASCII-8BIT, 63
asin, 296
asinh, 296
assoc, 311
at, 307
at_exit, 44
ataki SQL injection, 84
atan2, 296
atime, 323, 324
atrybuty, 202, 205
attr_accessor, 206, 213, 256, 267
attr_reader, 206, 213, 256, 267
attr_writer, 206
attribute, 268
attributes, 268
autoload, 240
automatyczne ładowanie modułów, 240
awarie, 367

B

backtrace, 150
Base64, 232, 233
Base64.helper, 234
Base64::Decoder, 234
Base64::Encoder, 234
basename, 321

BasicObject, 75, 221, 266
BEGIN, 44, 158
between?, 234
bezpieczeństwo, 356, 374
 wątki, 263
białe znaki, 37
BigDecimal, 46, 296
Bignum, 15, 46, 48
BINARY, 63
bind, 337
binding, 190
Binding, 190, 251
blok, 13, 40, 136
 parametry, 137
 przekazywanie argumentów, 139
 przypisanie równolegle, 139
 składnia, 136
 wartość, 137
 zasięg zmiennych, 138
BOM, 42
brakujące metody, 264
brakujące stałe, 264
break, 38, 127, 143, 262
 blok, 187
 lambda, 187
 obiekty proc, 187
 z wartością, 143
broadcast, 352
bytesize, 280

C

call, 164, 184, 191
callcc, 164
caller, 260
capitalize, 281
capitalize!, 281
Cardinal, 21
case, 121
 ==, 122
case equality, 17, 78, 104
casecmp, 281
catch, 147
cbrt, 296
ceil, 295
center, 282
chdir, 321, 323
chmod, 325

choice, 307
chomp, 282
chomp!, 282
chop, 282
chop!, 282
ciało, 40
ciąg Fibonacciego, 161
clas_attrs, 268
class, 169, 203, 248
Class, 36, 78
Class#new, 227
Class::new, 227
class_attrs, 268
class_eval, 251, 267, 268
 definiowanie metod, 267
class_exec, 252
class_variables, 252
classify, 320
ClassTrace, 270
clear, 309
clone, 83, 228
close_read, 334
close_write, 334
closed?, 335
closure, 166, 188
CMath.sqrt, 297
coerce, 82
collect, 302
collect!, 309
compact, 309
compact!, 309
Comparable, 79, 234
compile, 286
Complex, 46, 80, 297
Complex.polar, 297
concat, 67, 280
ConditionVariable, 352, 353
ConditionVariable.new, 352
conject, 352
conmap, 348
const_defined?, 253
const_get, 253
const_missing, 259, 264
constant_missing, 264
constants, 252
continuation, 164
Continuation, 164
copy_stream, 324, 330

cos, 296
count, 283, 306
cover?, 71
CRLF, 52
crypt, 283
cudzysłowy podwójne, 51
cudzysłowy pojedyncze, 50
cycle, 301, 309
cykl życia wątku, 342
czas istnienia obiektu, 74
częściowa aplikacja funkcji, 195

D

dane
 niebezpieczne, 374
 niepewne, 374
 wyjściowe, 21
data, 299
DATA, 328, 329, 361
datagramy, 337
day, 299
decode, 233
Decoder, 233
def, 15, 167
default, 315
define_finalizer, 261
define_method, 198, 255, 256, 268
 definiowanie metod, 268
defined?, 100, 110
definiowanie
 atrybutowe metody dostępu, 256
 klasy, 17, 202, 250
 klasy wyjątków, 150
 metody, 15, 167, 255, 267, 268
 metody singletonowe, 169
 metody za pomocą obiektów klasy
 Proc, 192
 moduły, 250
 operatorzy, 207
deklarowanie kodu, 158
delegacja, 266
delete, 283, 314, 319, 325
delete?, 319
delete_if, 314, 319
deq, 351, 352
deszeregowanie obiektów, 230
Dir, 321

Dir.delete, 325
Dir.entries, 322
Dir.foreach, 322
Dir.glob, 322
Dir.mkdir, 325
Dir.unlink, 325
Dir[], 322
directory?, 323, 324
dirname, 321
divide, 320
długość łańcuchów, 280
długość tablicy, 307
dokumentacja, 22
dokumenty
 miejscowe, 54
 osadzone, 32
dołączalne moduły przestrzeni nazw, 235
dołączanie elementów do łańcucha, 19
dołączanie plików, 41, 236
dołączenie, 102
Domain Specific Languages, 248
domieszki, 234
domknięcie, 166, 188
 wiązania, 190
 współdzielone zmienne, 189
domyślne kodowanie zewnętrzne, 43
domyślne wartości parametrów, 175
dopasowywanie wzorców, 290
dopełnienie, 102
dostęp do podłańcuchów, 59
dostęp do tablicy, 91, 209
dostęp do znaków, 59
dowiązanie, 251
dowiązanie symboliczne, 325
downcase, 281
downto, 132
drop, 305
drop_while, 305
DRY, 215
DSL, 248, 274
duck punching, 269
duck typing, 76
dump, 283
dup, 83, 228
dynamiczna modyfikacja metod, 269
dynamiczne tworzenie metod, 267
działania arytmetyczne na czasie, 300

dziedziczenie, 220
metody, 221
metody klasowe, 224
stałe, 226
zmienne egzemplarza, 225
zmienne klasowe, 225
dzielenie, 48
całkowitoliczbowe, 48

E

each, 13, 18, 127, 129, 160, 209, 214, 309, 319, 330
each_byte, 132, 282
each_char, 62, 132, 282
each_cons, 301
each_in_snapshot, 136
each_index, 309
each_key, 314
each_line, 132, 282
each_object, 261
each_pair, 214, 314
each_slice, 301
each_value, 314
each_with_index, 301, 309
efekty uboczne przypisania, 92
egzemplarz klasy, 203
eigenclass, 199, 231, 240, 241
element, 276
else, 117, 155, 168
elsif, 117
embedded document, 32
empty?, 280, 307, 314, 318
encode, 63, 233, 285
encode!, 285
Encoder, 233
encoding, 62
Encoding, 43, 63, 64, 65
Encoding.aliases, 65
Encoding.default_external, 43, 65
Encoding.find, 65
Encoding.locale_charset, 43
Encoding.name_list, 65
Encoding::BINAR, 329
end, 116, 117, 203
END, 158
end_with?, 281
enq, 351, 352

ensure, 156, 168, 350
enum_for, 132, 302
Enumerable, 129, 131, 132, 193, 234, 235, 301, 302, 303, 314
Enumerable.average, 198
Enumerable::Enumerator, 131, 302
enumeracja, 209
enumerator, 131, 302
enumeratory, 131, 302
ENV, 362, 374
eof?, 334, 335
EOFError, 153, 155, 220, 331
eql?, 69, 78, 210, 280
equal?, 77, 221
erf, 296
erfc, 296
EUC, 42
EUC-JP, 294
eval, 250, 251, 253
even?, 295
every, 262
exception, 151
Exception, 80, 148, 149, 150, 220
Exception.backtrace, 260
exchange, 353
Exchanger, 353
executable?, 323, 324
exist?, 323
exit, 346, 373
exit!, 171
exp, 296
expand_path, 321
external_encoding, 328
extname, 321

F

fail, 373
false, 39, 73, 82, 86, 105, 346
FALSE, 362
fdiv, 48
fetch, 307, 313
fiber, 163
Fiber, 160, 163
Fiber.new, 160
Fiber.yield, 160, 161
File, 320, 321
File.copy_stream, 331

File.expand_path, 321
File.fnmatch, 322
File.foreach, 331
File.ftype, 323
File.identical?, 321
File.new, 326
File.open, 326, 371
File.read, 331
File.readlines, 331
File.stat, 324
File::ALT_SEPARATOR, 321
File::FNM_PATHNAME, 322
File::SEPARATOR, 321
File::Stat, 323
file?, 323, 324
fill, 309
find_all, 304
finite?, 295
first, 305, 307, 308
fixed_encoding?, 294
Fixnum, 15, 46, 47, 169
flatten, 312, 314
flip-flop, 107
Float, 48, 49
floor, 295
flush, 335
fnmatch, 322
force_encoding, 63, 285
for-in, 125
fork, 371, 372
formatowanie tekstu, 283
freeze, 19, 84, 257
frexp, 296
frozen?, 84
fsync, 335
FTP, 340, 348, 371
ftype, 323, 324
funkcje
 globalne, 90, 366
 globalne zdefiniowane przez
 użytkownika, 368
 konwertujące, 81, 367
 raportujące, 369
 refleksji, 367
 wejściowe, 369

G

gamma, 296
Gang of Four, 133
garbage collector, 75
GC, 261
GC.disable, 261
GC.enable, 261
GC.start, 261
gem, 22, 23
gem enviroment, 23
gem install, 22
gem list, 23
gem uninstall, 23
gem update, 23
generatory, 161
generowanie danych XML z walidacją, 276
getbyte, 331
getc, 331
getlocal, 299
gets, 84, 329, 330, 331, 369
getter, 94, 207
getutc, 299
getwd, 323
global_variables, 252
gm, 299
gmtime, 299
gniazda, 335
godzina, 299
goto, 164
grep, 304
group_by, 304
grupy wątków, 347
gsub, 281, 293
gsub!, 293

H

has_key?, 312
has_value?, 312
hash, 13, 69, 211
Hash, 13, 69, 81, 129, 211, 316
hash table, 69
hashcode, 69
hasz, 68
here document, 54
hermetyzacja zmiennych egzemplarza, 204

hierarchia klas, 220
klasy wyjątków, 149
hour, 299
HTTP, 340, 348, 371
hypot, 296

I

idempotentne wyrażenia, 92
identyfikator obiektu, 75
identyfikatory, 34
znaki interpunkcyjne, 35
znaki Unicode, 35
if, 116, 118
else, 117
elsif, 117
modyfikator, 118
wartość zwrotna, 118
IMAP, 340
implementacja generatorów, 161
implementacje języka Ruby, 20
include, 249
include?, 71, 249, 281, 303, 309, 310, 312
indeksowanie
 łańcuchy, 60
 tablice asocjacyjne, 312
index, 281, 293, 309, 312
IndexError, 134
infinite?, 295
Infinity, 48, 102
informacje o plikach, 323
inherited, 258
inicjalizacja obiektów, 203, 227
initialize, 74, 203, 216, 217, 224, 227
initialize_copy, 83, 228
inject, 193, 306
insert, 280
inspect, 64, 283, 311, 316, 320
instance_eval, 198, 199, 221, 251
instance_exec, 252
instance_method, 192, 197
instance_of?, 75, 210, 249
instance_variable_get, 268
instance_variables, 252
instrukcje, 14, 37
instrukcje warunkowe, 19, 116
Integer, 13, 46
integer?, 295

interaktywność, 21
intern, 72, 283
internal_encoding, 328
interpolacja łańcuchów, 14, 52
interpreter, 20
introspekcja, 248
IO, 81, 301, 325, 369
IO.copy_stream, 330
IO.foreach, 330
IO.popen, 341
IO.read, 330
IO.readlines, 330
IO-bound, 341
IOError, 153, 220
irb, 21
IronRuby, 20
is_a?, 76, 210, 235, 249
isdst, 299
iteracja
 kolekcje, 301
 łańcuchy, 61
 równoległa, 135
iteratory, 13, 124, 126, 130
 each, 18, 129, 209
 numeryczne, 127
 ponowne uruchomienie bieżącej
 iteracji, 145
przejście do następnej iteracji, 144
tablice asocjacyjne, 314
 wewnętrzne, 133
 współbieżne, 348
zbiory, 319
zewnętrzne, 133, 302

J

Java, 77
jawne przekazywanie obiektów klasy Proc, 179
jcode, 61, 65
język
 do wyspecjalizowanych zastosowań,
 248, 274
 DSL, 248, 274, 275
 Ruby, 12
 YAML, 84
join, 311, 321, 352
JRuby, 20

K

- kacze typowanie, 76, 207
 - równość, 210
- katalogi, 320
 - lista zawartości, 322
 - nazwy, 321
 - tworzenie, 324
 - usuwanie, 324
 - zmiana nazwy, 324
- Kernel, 36, 81, 147, 164, 235, 240, 248, 367, 369
- Kernel.__callee__, 260
- Kernel.__method__, 260
- Kernel.`, 370
- Kernel.at_exit, 373
- Kernel.caller, 260
- Kernel.eval, 250
- Kernel.exec, 341, 372
- Kernel.fork, 341
- Kernel.lambda, 182
- Kernel.load, 270
- Kernel.loop, 134, 302
- Kernel.open, 326, 348
- Kernel.proc, 182
- Kernel.rand, 298
- Kernel.require, 270
- Kernel.select, 338, 348
- Kernel.sprintf, 283
- Kernel.system, 341, 371
- Kernel.trace_var, 261
- Kernel.trap, 373
- Kernel.untrace_var, 261
- key?, 312
- keys, 314
- kind_of?, 76, 249
- klasowe zmienne egzemplarza, 217
- klasy, 17, 248, 360
 - abstrakcyjne, 223
 - akcesory, 205
 - anonimowe, 213
 - atrybuty, 205
 - BasicObject, 75, 221
 - definiowanie, 17, 202
 - dziedziczenie, 220
 - eigenklass, 240, 241
 - hermetyzacja zmiennych egzemplarza, 204
 - hierarchia, 220
 - metody klasowe, 215
- Module, 206, 232
- nadklasy, 220
- Object, 75, 220
- operatory, 207
- otwarte, 19
- Singleton, 231
- stałe, 216
- Struct, 213
- tworzenie, 203
- tworzenie egzemplarza, 203
- zmienne klasowe, 216
- klasy wyjątków, 149
- klient, 335, 338
- klucz, 68
- klucz sortowania, 303
- kod mieszający, 69, 211, 316
- kod źródłowy, 41
- kodowanie, 41, 64, 328
 - ASCII, 63
 - BINARY, 63
 - łańcuchy, 61
 - wyrażenia regularne, 294
 - źródło, 43
- koercja typów przez operatory arytmetyczne, 82
- kolejki, 310, 351, 352
- kolekcje, 300
 - enumeratory, 302
 - iteracja, 301
 - iteratory zewnętrzne, 302
 - konwersja, 301
 - pobieranie fragmentów, 304
 - przeszukiwanie, 303
 - redukowanie, 305
 - sortowanie, 303
- kombinacje zbiorów, 317
- komentarze, 32
 - dokumentacyjne, 33
 - shebang, 40
 - wielowierszowe, 32
- komunikaty o błędach, 259
- koniec instrukcji, 37
- kontrola przepływu sterowania, 141
- kontynuacje, 159, 164
- konwersja
 - jawna, 80
 - kolekcje, 301
 - łańcuchy na symbole, 282

niejawna, 80
obiekty, 80
typy logiczne, 82
kopiowanie obiektów, 83, 228
krotność obiektów klasy Proc, 184
krotność operatora, 99

L

lambda, 166, 181, 182
last, 307, 308
ldexp, 296
length, 59, 66, 280, 307, 314, 351
lgamma, 296
liczby, 46, 295
 arytmetyka, 48
 błąd zaokrąglania, 49
 dzielenie, 48
 dziesiętne, 296
 hierarchia klas, 46
 literały liczb całkowitych, 47
 literały liczb zmiennoprzecinkowych, 47
losowe, 298
modulo, 48
rzeczywiste, 46
ujemne, 48
wymierne, 46, 297
zaokrąglanie, 295
zespolone, 297
złożone, 46
zmiennoprzecinkowe, 46, 49
link, 325
lista metod, 254
lista zawartości katalogów, 322
listy wątków, 347
literały, 34, 86
 haszowe, 68
 lambda, 182
 liczby całkowite, 47
 liczby zmiennoprzecinkowe, 47
łańcuchowe, 50
łańcuchowe w cudzysłowach
 podwójnych, 51
łańcuchowe w cudzysłowach
 pojedynczych, 50
słowa kluczowe, 86
tablicowe, 66
wyrażenia regularne, 285

znakowe, 57
ljust, 282
load, 236, 237, 239
local, 299
local_variables, 252
LocalJumpError, 143, 186
localtime, 299
lock, 350, 352
log, 296
log10, 296
log2, 296
lstrip, 282
lvalue, 91
l-wartość, 91

Ł

ładowanie
moduły, 240
strony internetowe, 340
łańcuchowa reprezentacja obiektu, 205
łańcuchy, 50, 223, 280
 długość, 280
 dokumenty miejscowe, 54
 dostęp do podłańcuchów, 59
 dostęp do znaków, 59
 Encoding, 64
 formatowanie, 283
 indeksowanie, 60
 interpolacja, 52, 58
 iteracja, 61
 iterowanie, 282
 kodowanie, 61, 64, 285
 konwersja na symbole, 282
 literały, 50
 literały w cudzysłowach podwójnych, 51
 literały w cudzysłowach pojedynczych, 50
 literały znakowe, 57
 łączenie, 57
 modyfikacja, 56
 odwracanie, 283
 odwrotne apostrofy, 56
 ograniczniki literałów łańcuchowych, 54
 operatory, 57
 pakowanie, 284
 pobieranie podłańcuchów, 59
 podmiana podłańcucha, 60
 Range, 60

łańcuchy
 rozpakowywanie, 284
 sprawdzanie równości, 58
 String, 50, 281
 translacja zestawu znaków, 283
 usuwanie białych znaków, 281
 wstawianie znaków Unicode, 52
wyrażenia regularne, 60
zmiana kodowania, 63
zmiana wielkości liter, 281
znaki, 57
znaki wielobajtowe, 61
zwiększenie, 283
łańcuchy aliasów, 269
 bezpieczeństwo wątków, 271
 tworzenie na potrzeby śledzenia, 273
łączenie
 funkcje, 194
 łańcuchy, 57
łączność operatora, 99

M

macierze, 298
main, 368
manipulowanie tekstem, 366
map, 68, 127, 193, 302, 309
map!, 309
mapowanie argumentów na parametry, 177
Marshal.dump, 83
Marshal.load, 83
marshal_dump, 229, 230
marshal_load, 229, 230
marshaling, 83
match, 281
MatchData, 290
matematyka, 295
Math, 15, 296
mathn, 297
matrix, 298
Matrix, 298
Matsumoto, Yukihiro, 12
Matz, 12
max, 305
member?, 71, 303, 312
memoization, 196
merge, 313
merge!, 313

message, 90, 150
metaklasa, 241
metaprogramowanie, 191, 206, 248
method, 191
Method, 166, 191, 192, 194, 196
method_added, 259, 271
method_defined?, 254
method_missing, 257, 259, 264, 265, 266, 274, 276
 wysyłanie danych w formacie XML, 274
method_undefined, 259
metody, 15, 39, 166, 254
 [], 209
 after, 262
 aliasy, 171
 allocate, 227
 argumenty, 174
 argumenty blokowe, 178
 attr_accessor, 206
 attr_reader, 206
 attr_writer, 206
 chromione, 218
 class_eval, 251, 267
 class_exec, 252
 clone, 228
 const_missing, 264
 define_method, 268
 definiowanie, 15, 167
 definiowanie za pomocą obiektów
 klasy Proc, 192
 domyślne wartości parametrów, 175
 dostępowe, 90, 94
 dup, 228
 dziedziczenie, 221
 eql?, 78
 equal?, 77
 eval, 251
 every, 262
 fabryczne, 227
 freeze, 84
 initialize, 203, 216, 227
 initialize_copy, 228
 instance_eval, 251
 instance_exec, 252
 iteracyjne, 130
 logarytmiczne, 296
 mapowanie argumentów na parametry, 177
 marshal_dump, 229, 230

marshal_load, 229, 230
method, 191
method_missing, 265
modyfikujące, 170
mutatory, 170
nawiasy, 172
nazwy, 170
new, 203, 227
nieustalona liczba argumentów, 176
obowiązkowe używanie nawiasów, 173
obsługa wyjątków, 168
określanie widoczności, 219
opcjonalne nawiasy, 172
operatorowe, 171
parametry, 15
private, 219
protected, 219
prywatne, 218
przekazywanie tablic, 176
przesłanianie, 222
public, 219
publiczne, 218
rozstrzyganie nazwy, 223
singletonowe, 15, 169, 231
synchronize, 263
system operacyjny, 367
tablice, 176
tablice asocjacyjne, 177
to_s, 205
trygonometryczne, 296
tworzenie dynamiczne, 267
ustawiające, 94, 206
usuwanie definicji, 169
wartość zwrotna, 167
widoczność, 218
wywołanie, 38, 89, 166
wywołanie w stylu funkcyjnym, 218
znak & w wywołaniu, 180
zwracanie wartości, 16, 141, 167
metody dostępu swobodnego, 334
metody klasowe, 215
 dziedziczenie, 224
metody obiektów wyjątków, 150
metody zwrotne, 258
min, 299, 305
minmax, 305
mixins, 234
Module, 36, 104, 206, 232, 240, 248, 254

Module.create_alias, 271
Module.synchronize_method, 271
module_function, 235, 236
modulo, 48
moduły, 17, 39, 232, 248, 249, 360
 automatyczne ładowanie, 240
 domieszki, 234
 Enumerable, 301
 funkcje, 236
 GC, 261
 Math, 296
 ObjectSpace, 261
 przestrzenie nazw, 232
 rozstrzyganie nazwy, 242
 singletonowe, 240
 wyszukiwanie, 242
 zagnieżdżanie, 233
modyfikacja
 elementy tablicy, 308
 literały łańcuchowe, 56
 punkty, 212
 współbieżna, 136
modyfikatory, 112
 if, 118
 rescue, 157
 until, 125
 while, 125
monkey chaining, 269
month, 299
MRI, 20
mtime, 323
multiplier, 189
multipliers, 190
mutatory, 16, 170
mutex, 263
Mutex, 263, 350, 353

N

nadawanie nazwy obiektowi wyjątku, 152
nadklasy, 220
nadzbiór, 317
nadzbiór właściwy, 317
named backreference, 294
NameError, 88
NaN, 48, 105
nan?, 295
nawiasy, 172
 przypisanie równolegle, 98

nazwy, 213
katalogi, 321
metody, 170
pliki, 321
zmienne, 17
Net::HTTP, 340
new, 74, 203, 227, 286, 325, 326
next, 133, 144, 162, 187
wartość bloku, 145
niestandardowe struktury sterujące, 262
nieustalona liczba argumentów, 176
nil, 19, 39, 73, 82, 86, 103, 105, 346
NIL, 362
NoMemoryError, 155
nonzero?, 295
not, 100, 105
NOT, 102, 103
now, 299
Numeric, 46, 78, 101, 295

O

obiektowość, 12
obiekty, 12, 73, 202
akcesory, 205
atrybuty, 205
czas istnienia, 74
funkcje konwertujące, 81
identyfikator, 75
inicjalizacja, 203, 227
konwersja jawnia, 80
konwersja niejawnia, 80
kopiowanie, 83, 228
lambda, 181
porównywanie, 77
porządkowanie, 79
proc, 181
przeliczalne, 126, 129
referencje, 73
self, 204, 207
serializacja, 229
sprawdzanie klasy, 75
synchronizowane przez delegację, 266
szeregowanie, 83
tożsamość, 75
tworzenie, 203, 227
wartości bezpośrednie, 74
zamrażanie, 84

zanieczyszczanie, 84
zapisywanie stanu, 83
Object, 15, 36, 75, 128, 220, 248, 254
Object.bindings, 251
Object.extend, 250
Object.inherited, 270
Object.mutex, 266, 271
Object.object_id, 261
Object.trace, 265
object_id, 69, 75
ObjectSpace, 261
ObjectSpace._id2ref, 261
ObjectSpace.define_finalizer, 261
ObjectSpace.garbage_collect, 261
ObjectSpace.undefined_finalizer, 261
obliczenia, 39
obsługa niezdefiniowanych metod, 257
obsługa wyjątków, 148, 152
metody, 168
według typu, 153
wyjątki, 154
odbiorca, 166
odczyt ze strumienia, 329
bajty, 331
całe pliki, 330
wiersze danych, 329
znaki, 331
odd?, 295
odwołania do nazw metod, 72
odwołania do stałych, 88
odwołania do zmiennych, 87
odwracanie łańcucha, 283
odwrotne apostrofy, 56
ograniczniki literalów łańcuchowych, 54
ograniczone środowisko wykonawcze, 374, 376
określanie kodowania programu, 42
open, 324, 325, 326, 372
open-uri, 340, 348
operacje logiczne, 102
operatory, 14, 39, 86, 99
 $<=>$, 79
 $=~$, 79
 $==$, 77
 $==$, 17, 78
 arytmetyczne, 48, 82, 101
 bitowe, 49
 defined?, 110
 definiowanie, 207

dołaczanie, 102
krotność, 99
logiczne, 105
łańcuchowe, 57
łączność, 99, 100
metodowe, 207
porównywanie, 103
potęgowanie, 101
priorytety, 99, 100
przesunięcie, 102
przypisanie, 16, 110
równość, 104
równość case, 78, 122
równość przypadków, 17
równość szczególna, 104
splat, 97
warunkowy, 109
opóźnianie wykonania, 262
opróżnianie strumieni, 334
or, 100, 105
OR, 102, 103
ostrzeżenia, 367
otwieranie
 pliki, 326
 strumienie, 325

P

pack, 311
pakiety, 22
pakowanie ładowanych plików, 239
pakowanie łańcuchów binarnych, 284
parametry blokowe, 137, 138
parametry metod, 15
partial application, 195
partition, 281, 304
patchlevel, 362
PATH, 374
pętle, 19, 124
 for-in, 125
 ponowne uruchomienie bieżącej iteracji, 145
 przejście do następnej iteracji, 144
 przerwanie wykonywania, 143
 until, 124
 while, 124
pipe, 325
planowanie wykonywania wątków, 344

pliki, 320
dowiązanie symboliczne, 325
informacje, 323
nazwy, 321
odczyt, 330
odczyt współbieżny, 347
otwieranie, 326
tryb, 326
tworzenie, 324
usuwanie, 324
zmiana nazwy, 324
pobieranie fragmentów kolekcji, 304
podklasy, 220
podtablice kolekcji, 301
podzbiór, 317
podzbiór właściwy, 317
polecenia systemu operacyjnego, 370
połączenia sieciowe, 335
ponowne uruchomienie bieżącej iteracji
 pętli, 145
pop, 310
POP, 340
popen, 325
porównywanie, 103
 obiekty, 77
 obiekty klasy Proc, 185
punkty, 209
tablice, 309
wartości, 79
porządek w klasach, 79
porządkowanie
 obiekty, 79
 punkty, 212
pos, 334
potęgowanie, 49, 101
potomkowie klasy, 220
powłoka Ruby, 21
powtarzanie wykonania, 262
poziomy zabezpieczeń, 374, 375
predefiniowane funkcje globalne, 366
primary expression, 39
print, 333, 369
printf, 369, 370
priority, 345
priorytety operatorów, 99, 100
priorytety wątków, 345
private, 218
private_class_method, 258

proc, 166, 181
Proc, 166, 179, 184, 194, 196, 352
definiowanie metod, 192
krotność obiektów, 184
porównywanie obiektów, 185
tworzenie obiektów, 181
Proc.new, 182, 187
Process, 341
Process.exit, 373
Process.exit!, 373
Process.fork, 371
procesy, 372
program, 26, 32
uruchamianie, 43
wielowątkowy, 263, 340, 347
zachłanny, 341
programowanie
funkcyjne, 192
sieciowe, 335
współbieżne, 351
propagacja wyjątków, 153
proper_subset?, 318
proper_superset?, 318
protected, 218
przechwytywanie sygnałów, 373
przecięcie, 102
przecinek, 112
przedrostki, 16
przeglądarka dokumentacji, 22
przekazywanie argumentów, 74
do bloku, 139
do lambd, 188
do obiektów proc, 188
tablice, 176
przepływ sterowania, 141
przerzutniki, 107
logiczne, 107
przesłanianie metod, 222
metody prywatne, 223
przestrzenie nazw, 232
przesunięcie, 102
przeszukiwanie
kolekcje, 303
tablice, 309
przodkowie klasy, 220, 249
przypisanie, 16, 91, 110
do atrybutów, 93
do elementów tablicy, 93
do stałych, 93
do tablicy, 91
do zmiennych, 92
skrócone, 91, 94
przypisanie równoległe, 94, 96
jedna wartość po lewej i kilka wartości
po prawej stronie, 96
kilka wartości po lewej, jedna tablica
po prawej stronie, 96
nawiasy, 98
różne liczby wartości po obu stronach, 97
splat, 97
taka sama liczba wartości po lewej
i prawej stronie, 96
wartość zwrotna, 99
wywołanie metod, 98
przyrostki, 16
public, 218
public_class_method, 258
public_method, 191
punkty kodowe Unicode, 264
push, 310, 351
putc, 333, 369
puts, 21, 333, 369

Q

Queue, 351, 352

R

rails, 293
raise, 147, 148, 150, 373
rand, 298
Range, 17, 19, 60, 70, 129, 301
raportowanie, 368
rassoc, 311, 312
Rational, 46, 48, 80, 82
read, 332
read_noblock, 332
read_nonblock, 338
readable?, 323, 324
readbyte, 331
readbytes, 332
readchar, 331
readline, 329, 330, 331, 369
readlines, 369
readpartial, 332, 338

readwrite, 268
recvfrom, 337
redo, 145
reduce, 194, 306
redukowanie kolekcji, 305
referencje, 73
refleksja, 191, 248, 367
aliasymetod, 255
class_eval, 251
class_exec, 252
define_method, 256
definiowanie klas, 250
dowiązanie, 251
eval, 251
instance_eval, 251
instance_exec, 252
klasy, 248
lista metod, 254
metody, 248, 254
metody zwrotne, 258
moduły, 248
obsługa niezdefiniowanych metod, 257
przodkowie, 249
stałe, 252
testowanie zmiennych, 252
tworzenie aliasów metod, 255
ustawianie widoczności metod, 257
ustawianie zmiennych, 252
wykonywanie bloków, 250
wykonywanie łańcuchów, 250
wywołanie metod, 255
zmienne, 252
regex, 285
regexp, 285
Regexp, 17, 81, 285, 286
Regexp.compile, 286
Regexp.escape, 286
Regexp.last_match, 290
Regexp.union, 286
reguły tworzenia identyfikatorów, 35
reject, 304
reject!, 314, 319
rename, 324
replace, 309, 313
require, 23, 236
require_relative, 237
rescue, 100, 112, 146, 147, 148, 152, 157, 168
 definicje metod, klas i modułów, 157
 modyfikator, 157
 retry, 155
 respond_to?, 76, 249, 254
 resume, 160, 163
 retry, 146
 rescue, 155
 return, 38, 127, 141, 167, 185
 blok, 185
 lambda, 185
 obiekta proc, 185
 reverse, 312
 reverse_each, 309
 rewind, 162, 334
 ri, 22
 rindex, 281, 309
 rjust, 282
 rmdir, 325
 round, 295
 rozgałęzianie procesów, 371
 rozmiar tablicy, 307
 rozpakowywanie łańcuchów binarnych, 284
 rozstrzyganie nazwy metody, 223, 242
 rozwijanie nazw metod, 170
 równość, 77, 104, 210
 równość przypadków, 17
 równość szczególna, 104
 rpartition, 281
 rstrip, 282
 Rubinius, 21
 ruby, 356
 opcje, 357, 360
 opcje informacji, 358
 opcje kodowania, 358
 opcje ostrzeżeń, 358
 opcje przetwarzania tekstu, 358
 Ruby, 12
 RUBY_PATCHLEVEL, 362
 RUBY_PLATFORM, 362
 RUBY_RELEASE_DATE, 362
 RUBY_VERSION, 362
 RubyGems, 22, 23, 238
 RUBYLIB, 375
 RUBYOPT, 23, 375
 run, 346
 rvalue, 91
 r-wartość, 91

S

sandbox, 356, 376
scalar?, 295
SCRIPT_LINES_, 260
Season, 230
sec, 299
seek, 334
select, 304, 339
self, 39, 86, 89, 191, 204, 207, 224, 249, 368
semantyka wywołań, 188
semantyka yield, 188
semicoroutines, 163
semiwpółprogramy, 160, 163
send, 197, 337
separatory, 112
sequence, 130
serializacja, 229
serwer, 336
 multipleksujący, 338
 wielowątkowy, 348
set, 317
Set, 316, 318
set_backtrace, 150
set_encoding, 328
setgid, 374
setter, 94, 207
setuid, 374
shebang, 40
shift, 136, 310, 315
side effect, 92
sieć, 335
 datagramy, 337
 gniazda, 335
 klient, 335, 338
 połączenia, 335
 serwer, 336
SIGINT, 373
singleton_method_undefined, 259
SIGUSR1, 373
SIGUSR2, 373
sin, 296
singleton, 15
Singleton, 231
singleton_method_added, 259
singleton_method_removed, 259
singleton_methods, 254
sinh, 296

size, 59, 66, 280, 307, 314, 323, 324, 351
size?, 323
SizedQueue, 351
SJIS, 42, 294
składnia, 14
 wyrażenia regularne, 287
skrócone przypisanie, 94, 95
skróty w skryptach jednowierszowych, 370
sleep, 335, 346
slice, 280, 308
slice!, 280
słabe referencje, 261
słowa kluczowe, 36, 86
 alias, 171
 break, 143
 case, 121
 class, 203
 def, 167
 if, 116, 118
 next, 144
 redo, 145
 rescue, 157
 retry, 146
 return, 141, 167, 185
 undef, 169
 unless, 120
 until, 124
 while, 124
 yield, 131
SMTP, 340
sort, 303, 309
sort!, 309
sort_by, 303
SortedSet, 316, 317
sortowanie
 kolekcje, 303
 tablice, 309
spacje, 38
spamiętywanie funkcji, 196
splat, 97
split, 281, 321
sprawdzanie
 bezpieczeństwo, 374
 klasa obiektu, 75
 przynależność do zakresu, 71
 stan wątków, 346
 typy, 207
 zmienne, 252

sprintf, 52, 102, 370
SQL injection, 84
sqrt, 296
squeeze, 283
rand, 298
stałe, 88, 216, 252, 361
 przypisanie, 93
 punkty kodowe Unicode, 264
stan obiektu, 17
StandardError, 134, 149, 153, 220
stany wątków, 345
start_with?, 281
status, 346
STDERR, 327, 329, 333, 362
STDIN, 327, 329, 362, 369
STDOUT, 327, 329, 333, 362
step, 132
stop?, 346
StopIteration, 133, 134, 302
store, 313
stos, 310
stos wywołań, 260
strefy czasowe, 299
strftime, 299, 300
String, 19, 50, 81, 281
String.chomp!, 330
String.new, 57
String.unpack, 284
stringio, 327
StringIO, 76
strip, 282
Struct, 213
Struct.new, 213
struktura bloku, 40
struktura leksykalna, 32
struktura plików, 40
struktura programu, 32
struktura syntaktyczna, 39
strukturny, 214
strumienie, 325
 ARGF, 327
 ARGV, 327
 DATA, 328
 dostęp swobodny, 334
 kodowanie, 328
 metody dostępu swobodnego, 334
 odczyt, 329
 opróżnianie, 334
 otwieranie, 325
 predefiniowane, 327
 testowanie, 334
 zamykanie, 334
 zapis, 333
sub, 281, 293
sub!, 281, 293
subset?, 317, 318
subtract, 319
succ, 70, 71, 283
sudoku, 25
sum, 283
suma, 102
sumy kontrolne, 283
sunday?, 299
superclass, 248
superset?, 318
swapcase, 281
switch, 123
sygnały, 373
Symbol, 14, 72, 169, 196
Symbol.to_proc, 197
symbole, 72
 odwołania do nazw metod, 72
symlink, 325
symlink?, 323
sync, 335
synchronizacja bloków, 263
synchronize, 263, 350
synchronize_method, 271
synchronized, 271
SynchronizedObject, 271
sysread, 332
sysseek, 334
system, 372
system przywracania pamięci, 261
system usuwający nieużywane obiekty, 74
SystemCallException, 334
SystemExit, 373
syswrite, 333
szeregowanie obiektów, 83, 230

Ś

ścieżka wczytywania, 237
śledzenie
 przypisania do zmiennych globalnych, 261
wykonywanie programu, 259

- śledzenie
wywołania metod, 265
załadowane pliki, 270
zdefiniowane klasy, 270
średnik, 112
środowisko, 356, 360
- T**
- tablica mieszająca, 69
tablice, 13, 66
concat, 67
dodawanie elementów, 67, 308
dostęp do elementów, 209
elementy, 307
indeksowanie, 67
indeksowanie podtablic, 307
iteracja, 309
jednowymiarowe, 306
kolejka, 310
konwersja na łańcuchy, 311
length, 66
liczba elementów, 66
modyfikacja elementów, 308
pobieranie wartości elementu, 66
porównywanie, 309
przecięcie zbiorów, 68
przeszukiwanie, 309
rozmiar, 307
size, 66
sortowanie, 309
stos, 310
suma zbiorów, 68
tworzenie, 66, 307
usuwanie elementów, 308
wstawianie elementów, 308
zbiory, 310
zwielokrotnianie, 67
tablice asocjacyjne, 13, 68, 311, 312
indeksowanie, 312
iteratory, 314
klucz, 68
klucze modyfikowalne, 69, 316
kod mieszający, 69, 316
konwersja na tablice jednowymiarowe, 314
literały haszowe, 68
porównywanie kluczy, 69
równość, 69
równość kluczy, 316
sprawdzanie przynależności
elementów, 312
tworzenie, 69, 312
usuwanie elementów, 313
wartości domyślne, 315
zamiana miejscami kluczy i wartości, 316
zapisywanie kluczy i wartości, 313
taint, 84, 374
tainted?, 84, 374
take, 305
take_while, 305
tan, 296
TCPServer, 335, 336, 337, 339
TCPServer.open, 336
TCPSocket, 335, 336, 337, 339
TCPSocket.new, 335
TCPSocket.open, 335
tekst, 50
tell, 334
terminate, 346
testowanie
strumienie, 334
zmienne, 252
Thread, 262, 341, 342, 346, 347
Thread.abort_on_exception, 343
Thread.current, 342
Thread.exit, 346
Thread.fork, 342
Thread.list, 347
Thread.main, 342
Thread.new, 262, 341, 342
Thread.new., 341
Thread.pass, 345
Thread.sleep, 346
Thread.start, 342
Thread.stop, 346
ThreadGroup, 347
ThreadGroup.list, 347
throw, 147, 164
Time, 299
times, 127, 132
to_a, 80, 81, 302
to_ary, 81, 96
to_b, 82
to_c, 80
to_enum, 132, 163, 302
to_f, 80, 81

to_hash, 81
to_i, 80, 81, 295
to_int, 81
to_proc, 196
to_r, 80
to_s, 21, 80, 82, 205, 311, 369
to_set, 317
to_splat, 303
to_str, 81
to_sym, 72, 283
tokens, 32
TOLEVEL_BINDING, 362
tożsamość obiektów, 75
tr, 283
tr_s, 283
trace!, 273
TracedObject, 265
transfer, 163, 164
transpose, 312
true, 19, 39, 73, 82, 86, 105
TRUE, 362
truncate, 295
tryb pliku, 326
tty?, 335
tworzenie
 aliasy, 172, 255
 długie komentarze, 32
 egzemplarz klasy, 203
 grupy wątków, 347
 identyfikatory, 35
 iteratory, 130
 katalogi, 324
 klasy, 203
 lista metod, 254
 lista wątków, 347
 lista zawartości katalogów, 322
 łańcuchy aliasów, 256, 269
 łańcuchy metod dla bezpieczeństwa
 wątków, 271
 łańcuchy metod na potrzeby śledzenia, 273
 obiekty, 227
 obiekty klasy Method, 254
 obiekty klasy Proc, 181
 obiekty wyjątków, 150
 pliki, 324
 podklasy, 220, 221
 podtablice kolekcji, 301
 tablice, 66, 307

tablice asocjacyjne, 312
wątki, 262, 341
zbiory, 317
TypeError, 153, 155
typy danych, 46, 248
 liczby, 46
 łańcuchy, 50
 symbole, 72
 tablice, 66
 tablice asocjacyjne, 68
 zakresy, 70

U

UCS, 64
UDP, 337
UDPServer, 337
UDPSocket, 337
unbind, 192
UnboundMethod, 192
undef, 169, 257
Unicode, 64
uniq, 310
uniq!, 310
Universal Character Set, 64
unless, 120
unlink, 325
unlock, 350
untaint, 374
until, 124
 modyfikatory, 125
untrace!, 273
upcase, 281
upcase!, 281, 309
update, 313
upto, 127, 132, 283
uruchamianie
 interpreter Ruby, 356
 program, 43
US-ASCII, 63
usec, 299
ustawianie widoczności metod, 257
ustawianie zmiennych, 252
ustawienia globalne, 363
usuwanie
 definicja metody, 169, 255
 katalogi, 324
 nieużywane obiekty, 75
 pliki, 324

utc, 299
utc?, 299
utc_offset, 299
UTF-8, 42, 62, 264, 294
Util, 240
utime, 325

V

valid_encoding?, 63
value?, 312
values, 314
values_at, 308, 313
Vector, 298

W

wait, 352
wakeup, 346
wartości, 14
 bezpośrednie, 74
wartość bloku, 137
wartość przypisania równoległego, 99
wątki, 159, 262, 340
 anulowanie, 346
 bezpieczeństwo, 263
 blokowanie, 352
 budzenie, 346
 cykl życia, 342
 główny wątek, 342
 grupy, 347
 iteratory, 348
 kolejki, 351, 352
 listy, 347
 mechanizm blokujący, 350
 Mutex, 350
 nieobsłużone wyjątki, 342
 odczyt plików, 347
 planowanie wykonywania, 344
 priorytety, 345
 serwer, 348
 sprawdzanie stanu, 346
 stany, 345
 synchronizacja bloków, 263
 synchronize, 263
 tworzenie, 341
 uruchamianie, 341
 uśpienie, 346

uśpiony, 345
wstrzymywanie, 346
wykluczanie, 349
wykonywalny, 345
wywłaszczanie, 345
zakleszczenia, 349, 350
zależność od platformy, 341
zamykanie, 346
zmiana stanu, 346
zmienne, 343
zmienne lokalne, 344
zmienne prywatne, 343
zmienne warunku, 352
wczytywanie
 bajty, 331
 znaki, 331
wday, 299
WeakRef, 261
wejście, 325
wektory, 298
while, 14, 124
 modyfikatory, 125
wiązania, 190
wiązanie bloku kodu
 z wywołaniem metody, 14
widoczność metod, 218, 257
wielkość liter, 35
with_index, 303
włókna, 159, 160
 argumenty, 161
 implementacja generatorów, 161
 wartości zwrotne, 161
 zaawansowane własności, 163
world_readable?, 323
world_writable?, 323
wprowadzanie tekstu, 366
writable?, 323, 324
write, 333
współbieżność, 340
współbieżność wątkowa, 159
współbieżny odczyt plików, 347
współdzielone zmienne, 189
współprogramy, 160
wstawianie znaków Unicode, 52
wycofywane funkcje ekstrakcji, 369
wyjątki, 146, 148, 260
 anulowanie propagacji, 156
 ArgumentError, 153
 backtrace, 150

definiowanie klas wyjątków, 150
else, 155
ensure, 156
exception, 151
Exception, 148, 149
klasy wyjątków, 149
message, 150
metody obiektów wyjątków, 150
nadawanie nazwy obiektowi wyjątku, 152
obiekty wyjątków, 149
obsługa, 148, 152
obsługa według typu, 153
propagacja, 153
raise, 148, 150
rescue, 152
retry, 155
set_backtrace, 150
StandardError, 153
tworzenie obiektów wyjątków, 150
wątki, 342
wyrzucanie, 148
zgłaszanie, 148, 150
zmienne globalne, 364
wyjście, 325
wykluczanie wątków, 349
wykonywanie
blok, 250
łańcuch, 250
program, 43
wczytany kod, 239
wykrzyknik, 16
wyłączanie systemu przywracania
pamięci, 261
wyprowadzanie tekstu, 366
wyrażenia, 14, 39, 86
wyrażenia pierwotne, 39
wyrażenia regularne, 17, 60, 285
dopasowywanie wzorców, 290
dopasowywanie wzorców
przy użyciu łańcuchów, 292
grupowanie, 287
klasy znaków, 287
kodowanie, 294
literały, 285
metody fabryczne klasy Regexp, 286
nazwane grupy, 287, 291
nazwane odwołania wsteczne, 294
odwołania wstecz, 287

podmienianie, 293
powtarzanie, 289
powtarzanie niezachłanne, 287
Regexp, 286
składnia, 287, 288
wyszukiwanie, 293
zakotwiczenia, 287, 289
zmienne globalne przechowujące dane
dopasowań, 292
znaki literalne, 287
znaki modyfikujące, 285
wyrzucanie wyjątków, 148
wysyłanie danych w formacie XML, 274
wyszukiwanie metod, 242
moduły klasowe, 243
wyszukiwanie stałych, 244
wyświetlanie danych wyjątkiowych, 21
wywieszczanie wątków, 345
wyołanie
funkcje globalne, 90
metody, 38, 89, 166, 255
metody na rzecz obiektów, 168
obiekty lambda, 184
obiekty proc, 184
polecenia systemu operacyjnego, 370
wzbogacanie zachowań, 223
wzorce projektowe, 133

X

XML, 274
XMLGrammar, 276
XMLGrammar.element, 276
XOR, 102

Y

YAML, 84
YARV, 20
yday, 299
year, 299
yield, 127, 130, 131, 140, 163, 180

Z

zagłędzanie
moduły, 233
przestrzeń nazw, 233

- zakleszczenia, 349, 350
zakresy, 17, 70, 107
 ciągłe, 70
 dyskretne, 70
 sprawdzanie przynależności, 71
 wartości brzegowe, 71
zamrażanie obiektów, 84
zamykanie, 367
zamykanie programu, 373
zamykanie strumieni, 334
zanieczyszczanie obiektów, 84
zaokrąglanie liczb, 49, 295
zapis w strumieniu, 333
zarządzanie pakietami, 22
zasada DRY, 215
zasięg zmiennych, 138
zbiory, 310, 316
 dodawanie elementów, 318
 iteratory, 319
 kombinacje, 317
 porównywanie, 317
 sprawdzanie istnienia wartości, 317
 tworzenie, 317
 uporządkowane, 316
 usuwanie elementów, 318
zero?, 295, 323
ZeroDivisionError, 48, 101, 155
zgłaszanie wyjątków, 148, 150
zip, 302
- zmiana
 nazwy katalogów, 324
 nazwy plików, 324
zmienne, 87, 252
 globalne, 16, 88, 362
 globalne dopasowywania wzorców, 365
 globalne obsługи wyjątków, 364
 globalne opcji wiersza poleceń, 366
 globalne przetwarzania tekstu, 364
 klasowe, 16, 87, 216
 lokalne, 88, 291
 lokalne wątków, 344
 niezainicjowane, 87
 obiektywe, 16, 87, 92
 prywatne wątków, 343
 przypisanie, 92
 zasięg, 138
znaki, 41, 57
 koniec instrukcji, 37
 nowy wiersz, 37
 Unicode, 35
znaki interpunkcyjne, 16, 34, 112
 identyfikatory, 35
znaki wielobajtowe, 61
 Ruby 1.8, 65
 Ruby 1.9, 61
zone, 299
zsynchronizowane bloki, 263

O autorach

David Flanagan jest programistą spędzającym większość czasu na pisaniu książek o językach programowania. Wśród jego publikacji wydanych przez wydawnictwo O'Reilly znajdują się *JavaScript: The Definitive Guide* i *Java in a Nutshell*. Flanagan z wykształcenia jest informatykiem i zdobył stopień inżyniera w Massachusetts Institute of Technology. Mieszka razem z żoną i dziećmi w regionie Pacific Northwest w USA, pomiędzy miastami Seattle w stanie Waszyngton a kanadyjskim Vancouver w Kolumbii Brytyjskiej.

Yukihiro Matsumoto („Matz”) jest twórcą języka Ruby i profesjonalnym programistą niegdyś pracującym dla japońskiej firmy open source’owej o nazwie netlab.jp. „Matz” jest także znany jako jeden z gorących zwolenników otwartego oprogramowania. Napisał kilka programów typu open source, do których zalicza się cmail — oparty na Emacs klient poczty w całości napisany w języku lisp Emacs. Ruby jest pierwszym jego produktem, który zyskał popularność poza Japonią.

Kolofon

Ptaki widoczne na okładce to kolibry rogatki (*Heliaactin bilophus*). Naturalnym miejscem występowania tych małych ptaków jest Ameryka Południowa, głównie regiony Brazylii i Boliwii. Preferują one suche otwarte przestrzenie, takie jak obszary trawiaste. Unikają gęstych i wilgotnych lasów.

Kolibry znane są z tego, że potrafią najszybciej ze wszystkich ptaków uderzać skrzydłami, a rogatek osiąga nawet do 90 uderzeń na sekundę (dla porównania najwolniejszy pod tym względem z ptaków — sęp — potrafi uderzać skrzydłami tylko jeden raz na sekundę). Dzięki tym zdolnościom i lekkiemu ciału kolibry mogą zawisać w powietrzu, utrzymując się na jednym poziomie dzięki szybkim uderzeniom skrzydeł. Potrafią także latać wstecz (są to jedyne ptaki, które posiadają tę umiejętność), co pozwala im utrzymać odpowiednią pozycję podczas zbiernania nektaru z kwiatów. Długi i cienki dziób umożliwia kolibrów sięganie daleko w głąb kwiatów. Co ciekawe, po portugalsku koliber nazywa się *beija-flor*, co oznacza „ptak, który całuje kwiaty”.

Po obu stronach głowy rogatka znajdują się kępki czerwonych, niebieskich i złotych piór. Grzbiet ma opalizujący kolor zielony, gardło i mostek są czarne, a brzuch biały. Ogon rogatka jest długi i spiczasty. Samica jest podobna do samca, ale nie posiada takiego samego wyróżniającego wzoru korony. Dzięki jaskrawym kolorom wcześni podróżnicy hiszpańscy nazwali te ptaki *joyas voladoras*, czyli „latające klejnoty”.

Istnieje wiele mitów związanych z kolibrami. W Brazylii czarny koliber jest zwiastunem śmierci w rodzinie. Starożytni Aztekowie czcili je, a ich duchowni usuwali złe zaklęcia za pomocą lasów pokrytych piórami kolibrów. Ponadto ptak ten jest symbolem zmartwychwstania, ponieważ Aztekowie wierzyli, że polegli wojownicy odradzali się jako kolibry. Aztecki bóg słońca i wojny Huitzilopochtli był reprezentowany przez kolibra, a jego imię można przetłumaczyć jako „Koliber z południa”, gdzie południe jest miejscem świata duchowego.

Rysunek na okładce pochodzi z książki *Animate Creation* J. G. Wooda. Krój czcionki użyty na przedniej okładce to ITC Garamond firmy Adobe. Tekst został napisany czcionką Linotype Birką, nagłówki krojem Adobe Myriad Condensed, a kod wydrukowano krojem TheSans Mono Condensed firmy LucasFonts.

PROGRAM PARTNERSKI

GRUPY WYDAWNICZEJ HELION



1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄZKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW
w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>